# Turbo-Pascal-Praxis

Teil 1

**Manfred Zander, Dresden**

*In diesem Heft beginnen wir mit einer Kursfolge, die – aufbauend auf dem Kurs Pascal (MP 9 und 11/87 sowie 3, 6, 9 und 11/88) – dem an Turbo-Pascal interessierten Programmierer Hilfsmittel und Werkzeuge für die Beherrschung dieser Sprache in die Hand geben soll, um effektiv programmieren zu können.*

## Einführung

Turbo-Pascal als eine schnell zu erlernende und doch sehr leistungsfähige Programmiersprache, hat sich in den letzten Jahren in vielen kleinen und mittleren CAD-Zentren unserer Betriebe einen der führenden Plätze erobert. Hier soll keineswegs versucht werden, ein Lehrbuch für diese Sprache vorzulegen, deshalb werden erste eigene Erfahrungen in dieser Sprache vorausgesetzt. Es geht darum, dem Programmierer kleine bewährte Mittel und Methoden vorzustellen, die universell in vielen Programmen genutzt werden können. Dem Anwender soll die Möglichkeit gegeben werden, sich bei der Programmierung voll auf seine Problemlösung zu konzentrieren. Die immer wiederkehrenden „kleinen" Probleme, wie Datenstrukturen, Bildschirmgestaltung, Dialogführung oder Druckerinitialisierung, die bei jeder Anwendung auftreten, belasten leider noch zu oft den Programmierer. Dabei müssen dann Dokumentationen gesichtet werden, Beschreibungen werden gesucht, und jede Lösung wird hundertemal neu gefunden. Dies alles kostet aber nicht nur Zeit. Durch die Nebenbeilösung der kleinen Probleme geraten diese uneffektiv, das wesentliche in den Programmen wird oft verdeckt, die Programme werden zu groß und zu langsam. Der Programmtext wird zu unübersichtlich, die angestrebte gute Strukturierung des Quelltextes leidet, ja es werden sogar Sprungbefehle notwendig. Die Folge all dessen ist natürlich auch eine steigende Fehlerhäufigkeit. Mit dem Vorstellen einiger kleiner Hilfsmittel, die sich bei mir bewährt haben, möchte ich meine Hilfe anbieten.

Von der Gestaltung des Dialoges Programm – Nutzer hängt ganz wesentlich ab, ob eine Software-Lösung später in der Praxis Fuß fassen wird oder ob die Arbeit des Programmierers umsonst war. Daher ist es notwendig, auf diesen Bereich sehr großen Wert zu legen. Programme, die im einfachen Rollmodus mit dem Nutzer kommunizieren, müssen der Vergangenheit angehören. Auch kleinere Lösungen müssen nach außen hin ansprechend und bedienfreundlich gestaltet sein. Selbst bösartige Fehleingaben dürfen nicht zu Programmabstürzen führen. Eine den jeweiligen Erfordernissen entsprechende Menütechnik sollte jedem Nutzer geboten werden. Um durch die Erfüllung dieser Forderungen die Programme nicht unnötig aufzubauschen und deren Lesbarkeit zu erhalten, ist es notwendig, einige Hilfsmittel zu nutzen und die Organisation des Dialoges von der speziellen Lösung zu trennen. Dann können diese Programmteile in einem Include-File versteckt werden und belasten den eigentlichen Programmtext nicht mehr.

## Eingabe und Ausgabe auf den Bildschirm

### Positioniertes Schreiben

Die wohl am häufigsten immer wieder aufeinanderfolgenden Befehle in Programmen sind gotoxy und write. Ganze Programmpassagen bestehen oft nur aus Kursorpositionierungen und Bildschirmausgaben. Es wird daher die Prozedur writeXY vorgestellt, die in allen Programmen zu Hause sein sollte. Mit ihr wird erst einmal nichts weiter erreicht, als im Programmtext diese zwei Befehle durch einen Prozedur-Aufruf zu ersetzen (Bild 1). Diese Prozedur kann den jeweiligen Bedürfnissen entsprechend modifiziert werden. Als Anregung dazu zwei Beispiele. In der Prozedur writeXYR wird neben der positionierten Ausgabe einer Zahl auf eine bestimmte Bildschirmposition gleich noch eine Standardformatierung dieser realisiert. Somit werden dann alle Zahlen konstant mit drei Nachkommastellen ausgegeben.

Ein Problem bei der Ausgabe von variablen Strings auf den Bildschirm ist das Überschreiben des Bildschirmrandes. Diese Problematik tritt immer dann auf, wenn Zeichenketten ausgegeben werden, die vorher vom Nutzer eingegeben wurden. Um davor sicher zu sein, bewährt sich die zentralisierte Lösung dieses Problems bei der String-Ausgabe, wie es in der Prozedur writeXYS vorgestellt wird.

### Positioniertes Lesen

Hier nun einige Hilfen für die Gestaltung von Eingabeanforderungen. Die Befehlskombination gotoxy mit read ist in Programmen häufig anzutreffen. Neben dieser Positionierung der Eingabe auf dem Bildschirm ist aber noch ein weiteres Problem zu lösen. Es soll eine absturzfeste Belegung von Variablen erreicht werden, so daß diesbezügliche Tests im Hauptprogramm entfallen können. Da diese Aufgaben für jede Eingabe gelten, ist wieder eine Zentralisierung der Eingabebehandlung günstig (Bild 2). Als erstes geben wir eine Funktion readXYS für die Eingabe von Strings an, in der natürlich noch keine Fehlerbehandlung notwendig ist.

```
function readXYS(x,y:integer):string80;
  var s:string80;
  begin                                  (* Lesen eines    *)
    gotoxy(x,y);read(s);                 (* Strings von    *)
    readXYS:=s                           (* Position x,y   *)
  end;

function readXYR(x,y:integer):real;
  var Io:integer;r:real;s:string80;
  begin
    repeat
      s:=readXYS(x,y);
      while pos(',',s)<>0 do s[pos(',',s)]:='.';
      while pos('+',s)<>0 do delete(s,pos('+',s),1);
      val(s,r,Io)
    until Io=0;
    readXYR:=r
  end;

function readXYI1(x,y:integer):integer;
  var Io,i:integer;s:string80;
  begin
    repeat                               (* Lesen einer    *)
      s:=readXYS(x,y);                   (* Integer-Zahl   *)
      val(s,i,Io)                        (* von Pos. x,y   *)
    until Io=0;                          (* bis Test Io    *)
    readXYI1:=i
  end;

function readXYI2(x,y:integer):integer;
  var r:real;
  begin
    repeat                         (* Lesen und Runden       *)
      r:=readXYR(x,y)              (* von Integer-Werten     *)
    until abs(r)<MaxInt;           (* Real-Eingabe erlaubt   *)
    readXYI2:=round(r)             (* bis Integer gueltig    *)
  end;
```

*Bild 2 Funktionen zum positionierten Lesen*

```
procedure writeXY(x,y:integer;s:string80);
  begin
    gotoxy(x,y);write(s)
  end;

procedure writeXYR(x,y:integer;r:real);
  begin
    gotoxy(x,y);
    write(r:10:3)           (* Formatierte Real-Ausgabe *)
  end;

procedure writeXYS(x,y:integer;s:string80);
  begin
    gotoxy(x,y);
    if length(s)>(80-x) then    (* Verhinderung des *)
      s:=copy(s,1,80-x);        (* Ueberschreibens  *)
    write(s)                    (* Bildschirmrand   *)
  end;
```

*Bild 1 Prozeduren zum positionierten Schreiben*

Für die Eingabe von Real-Werten ist aber eine Behandlung von falschen Eingaben notwendig. Zu schnell geben Nutzer z. B. statt des Punktes ein Komma ein und das Programm stürzt ab. Beim Erstellen einer diesbezüglichen Eingabefunktion ist nun zu entscheiden, ob falsche Eingaben nur zurückgewiesen werden oder ob eine automatische Fehlerkorrektur erfolgen soll. In der vorgestellten Funktion readXYR wird eine solche Korrektur realisiert. Sie liest zuerst einen String, in dem alle Zeichen zugelassen sind. Dann ersetzt sie eingegebene Kommas durch den vorgeschriebenen Punkt und löscht nicht zugelassene Pluszeichen, die ebenfalls zu Programmabstürzen führen würden. Erweist sich die Eingabe danach immer noch als falsch, wird die Eingabe erneut abgefordert.
Für das Einlesen von Integer-Werten bieten sich zwei Varianten an, die jede ihre speziellen Vor- und Nachteile hat. Deren Abwägung hängt von der konkreten Programmieraufgabe ab. In der Funktion readXYi1 wird die Eingabe solange wiederholt, bis die eingegebene Zeichenkette korrekt einer Integerzahl entspricht. Eine automatische Fehlerkorrektur, wie zum Beispiel das Löschen eines möglichen Plus-Zeichens, ist in ihr nicht vorgesehen. In der Funktion readXYi2 ist der Eingabespielraum sehr viel größer abgesteckt. Hier wird die volle Fehlerbehandlung der Funktion readXYR genutzt. Eingaben wie 3,0E10 werden akzeptiert. Da die Funktion readXYR mit Sicherheit eine gültige Realzahl übergibt, genügt ein zusätzlicher Test auf Zahlenbereichsüberschreitung für Integerwerte. Die Funktion readXYi2 übergibt an das Programm den ganzzahlig gerundeten Wert der eingegebenen Zahl.

### Kombiniertes Lesen und Schreiben

Durch die Nutzung der bis hier vorgestellten Prozeduren und Funktionen ist das Programm bereits vor einfachen Fehleingaben sicher. Für die Lesbarkeit des Programmtextes ist aber noch ungünstig, daß nicht sofort erkennbar ist, was nun eigentlich an der Position x, y gelesen werden soll, da die Ausgabe auf dem Bildschirm und die Abholung der Eingabe getrennt sind. Die oft in Programmtexten auftretenden Kombinationen der drei Befehle gotoxy, write und read sollten ebenfalls in Funktionen zentralisiert werden. Diese Funktionen realisieren zuerst die positionierte Ausgabe der Eingabeaufforderung auf dem Bildschirm. Direkt dahinter wird dann die absturzfeste Eingabe vom Nutzer entgegengenommen (Bild 3). Diese Aufgabe ist als Beispiel für die Belegung von Real-Variablen in der Funktion holeXYR gelöst.
Benötigt man in einem Programm nicht die volle Vielfalt der bisherigen Funktionen und Prozeduren, kann man die Wirkung der Funktion holeXYR, wie in der Funktion holeXYRsolo gezeigt, auch in sich selbst gewährleisten, ohne auf die Dienste vorangegangener Konstruktionen aufzubauen.
Die Nutzung dieser Funktion könnte dann zum Beispiel so aussehen:

**voli := holeXYR ('Innenvolumen:',10,5);**

In dieser komprimierten Befehlszeile ist nun auf einen Blick erkennbar, auf welcher Bildschirmposition welche Variable mit welcher Ausschrift eingelesen wird. Sie beinhaltet funktionell gleichzeitig die Realisierung von Zuweisung, Positionierung, Ausgabe und absturzfester Eingabe. Die einfache Variante dieser Funktion für den Rollmodus, mit dem Namen holeR, soll ebenfalls angegeben werden.
Mit dieser Funktion sieht die Eingabe der Maße eines Kreiszylinders und die Berechnung seines Volumens bereits so einfach aus:

```
Volum:=sqrt ( holeR ( 'Durchmesser' ))
*holeR('Hoehe')*pi/4.0;
```

Eine Zeile, in der einfachste Bedienerführung, Absturzfestigkeit, Lesbarkeit und hoher informativer Gehalt vereinigt sind. Als Veranschaulichung hier das gleiche einmal als einfache Realisierung ohne die hilfreiche Funktion:

```
repeat
  write ( 'Durchmesser : ');
  readln (s);
  val (s,d,lo)
until lo = 0;
repeat
  write ( 'Hoehe : ');
  readln (s);
val (s,h,lo)
until lo = 0;
Volum: = sqrt (d) *h*pi/4.0;
```

```
function holeXYR(tXt:string80;x,y:integer):real;
  begin                              (* Auf x,y Schreiben der *)
    writexy(x,y,tXt+' : '):          (* Eingabe-Aufforderung  *)
    holeXYR:=                        (* dahinter Lesen Real   *)
      readXYR(x+length(tXt)+3,y)
  end;


function holeXYRsolo(tXt:string80;x,y:integer):real;
  var s:string80;r:real;lo:integer;
  begin                              (* Auf x,y Schreiben der *)
    repeat                           (* Eingabe-Aufforderung  *)
      gotoxy(x,y);write(tXt+' : ');(* dahinter Lesen der     *)
      read(s);val(s,r,lo)            (* Real-Zahl bis Test lo *)
    until lo=0;
    holeXYR:=r
  end;


function holeR(tXt:string80):real;
  var s:string80;r:real;lo:integer;
  begin
    repeat
      write(tXt+' : ');               (* Eingabe-Aufforderung *)
      readln(s);val(s,r,lo)           (* auf aktuelle BS-Pos. *)
    until lo=0;                       (* Lesen und Zeilen-    *)
    holeR:=r                          (* schaltung bis lo     *)
  end;
```

***Bild 3** Funktionen zum kombinierten Lesen und Schreiben*

Die Vorteile selbst dieser kleinen Hilfsfunktion holeR dürften damit klar auf der Hand liegen, aber wir sind hiermit wieder in den Rollmodus abgeglitten, der für den Nutzer des Programmes unfreundlich ist. Dies würde sich vor allem bei falschen oder fehlerhaften Eingaben bemerkbar machen.

```
/---------------------------------------\
!Eingabe der Masze des Kreiszylinders:  !
!Durchmesser : 10,5                     !
!Durchmesser : 10.5                     !
!Hoehe : 20                             !
!Hoehe : 20                             !
!                                       !
\---------------------------------------/
```

***Bild 4** Bildschirmbild bei einfacher Realisierung*

Vertippte sich der Nutzer je einmal bei den beiden Eingaben, ergäbe sich dadurch das in Bild 4 dargestellte Bildschirmaussehen.
Nutzen wir die Funktion holeXYR, sieht die Zeile für die Volumenberechnung des Kreiszylinders so aus:

```
Volum: = sqrt (holeXYR('Durchmesser',10,5))
         *holeXYR('Hoehe',10,6)*pi/
4.0;
```

Die gleichen Vorteile sind erkennbar, aber die Lesbarkeit der Befehlszeile hat schon merklich abgenommen. Dafür haben wir jetzt aber erreicht, daß Fehleingaben das Bildschirmaussehen nicht mehr so stark verunstalten. Bei den gleichen falschen Eingaben hätten wir nach Abarbeitung der Befehlszeile den in Bild 5 dargestellten Bildschirm.

```
/---------------------------------------\
!                                       !
! Eingabe der Masze des Kreiszylinders: !
!                                       !
!                                       !
!           Durchmesser : 10.5          !
!           Hoehe : 20                  !
!                                       !
\---------------------------------------/
```

***Bild 5** Bildschirmbild bei Realisierung mit holeXYR*

Alle hier vorgestellten Prozeduren und Funktionen sind als Anregung gedacht. Sie können vom jeweiligen Nutzer beliebig modifiziert werden. Denkbar sind für die Lesefunktionen zum Beispiel noch das Vorlöschen der Eingabezeile oder auch das Vorgeben von Standard-Eingabewerten. Über die spezielle Form solcher Hilfen wird immer das konkret zu entwickelnde Programm entscheiden. Wesentlich ist die, durch die Anwendung von solchen Hilfsmitteln mögliche, erhebliche Verkürzung von Programmtexten sowie die Zentralisierung der Dialogprobleme. Eine gewissenhafte Beachtung bereits solcher Prinzipien kann sehr viele Schwierigkeiten vermeiden helfen.

### Das Auswahlmenü

Auswahlmenüs sind vor allem in dialog-

```
const                           (* Tasten-Code :       *)
  Nop         =   #00;          (* Quasileeres Char   *)
  Enter       =   #13;          (* Enter-Taste         *)
  Escape      =   #27;          (* Escape-Taste        *)
  kursortief=     #24;          (* Kursortaste tief   *)
  kursorhoch=     #05;          (* Kursortaste hoch   *)
                  (* Bildschirmsteuerzeichen : *)
  cleardown =  #$14;            (* Loeschen Bildschirm
                                   ab Kursorposition *)
  kursoron  =  #$82;            (* Kursor ein          *)
  kursoroff =  #$83;            (* Kursor aus          *)
  normvideo =  #$84;            (* normale Darstellung   *)
  invsvideo =  #$85;            (* inverse Darstellung   *)
  intsvideo =  #$86;            (* intensive Darstellung *)
  ivitvideo =  #$87;            (* invers und intensiv   *)
```

***Bild 6 Konstanten-Vereinbarungen für PC 1715***

orientierten Programmen die hauptsächliche optische Schnittstelle zwischen Programm und Nutzer. Der erste Bildschirmaufbau nach dem Start eines Programms bestimmt den sprichwörtlichen *ersten Eindruck*, aus dem oftmals sofort eine positive oder negative Grundeinstellung erwächst. Und dieser erste Bildschirmaufbau ist in den meisten Fällen das sogenannte Hauptmenü. Es sollte optisch ansprechend, schnell erfaßbar, sowohl leicht als auch komfortabel bedienbar und darüber hinaus auch noch sehr informativ sein. Programmtechnisch steht dem meist die einfache Aufgabe der Belegung einer Character-Variablen gegenüber. Wie wichtig diese Aufgabe aber von international führenden Softwareproduzenten genommen wird, zeigt die Computer- und Softwareentwicklung der letzten Jahre. Da kann der Nutzer Dateien mit einer Maus manipulieren und auf dem Bildschirm in einen Papierkorb fallen lassen. Ausgefeilte Menüs, beispielsweise mit Bildsymbolen, beherrschen den Markt. Die Realisierung dieser nahezu perfekten Lösungen ist aber nicht auf jeder Hardware möglich und vor allen Dingen auch gar nicht nötig. Vorgestellt werden soll daher eine einfache Lösung, die für den PC 1715 erstellt wurde, aber der Erfüllung der oben aufgestellten Forderungen schon sehr nahe kommt. Zur Demonstration der Lösung wird das Hauptmenü eines Spielprogramms genutzt.

Um den Quelltext des Beispielmenüs lesbarer zu gestalten, vereinbaren wir zuerst einige Konstantenbezeichner für die benötigten Tastaturcodes und Bildschirmsteuerzeichen (Bild 6). Diese Definitionen gelten selbstverständlich nur für den PC 1715 und ähnliche Rechner, zum Beispiel die mit einem entsprechenden Bildschirm ausgerüsteten Bürocomputer A 5120 und A 5130.

Entsprechend der programmtechnisch zu lösenden Aufgabe wird das komplette Menü in eine Funktion verpackt, die dem Programm nach seiner Abarbeitung einen gültigen Character-Wert übergibt. Damit wird erst einmal erreicht, daß der für den Aufbau des Bildschirmes zu schreibende Text im Hauptprogramm nicht stört. In dieser Funktion wollen wir nun organisieren, daß der Nutzer die einzelnen Positionen mit einem inversen Balken über Kursortasten anfahren und mit <Enter> auswählen kann.

Die allgemein übliche Möglichkeit der Auswahl mittels Kennbuchstaben bleibt dabei erhalten. Als zusätzlicher Service wird also erreicht, daß der Nutzer bei dieser Menüfunktion nun die Auswahl auf zwei unterschiedliche Arten treffen kann.

Gestaltet man nun den verbleibenden Platz auf dem Bildschirm noch ausreichend informativ und ansprechend, ist diese Ausbaustufe bereits ganz brauchbar und sicher auch für viele Aufgaben ausreichend. Wir wollen aber noch einen Schritt weiter gehen. Um dem Nutzer zu jeder Menüposition zusätzliche Informationen zu bieten, wird organisiert, daß zu jeder Balkenposition rechts neben dem Menü eine invers getastete kleine Tafel aufleuchtet, in der Zusatzinformationen enthalten sind.

Die hier vorgestellte programmtechnische Realisierung einer solchen Menüfunktion basiert auf einer konzentrierten Nutzung von Bildschirmsteuerzeichen. Die Einfassung der Bildschirmtexte, die später intensiv, invers oder invers-intensiv getastet werden sollen, in je zwei Bildschirm-Steuerzeichen normvideo, hat seinen Grund in der Tatsache, daß diese Steuerzeichen im Bildwiederholspeicher je ein Byte Platz beanspruchen. Beachtet man dies nicht, so würde sich der Text auf dem Bildschirm bei den ersten auszuführenden Umtastungen verschieben. Gleichzeitig muß organisiert werden, daß vor Sendung von invsvideo, intsvideo oder ivitvideo an den Bildschirm das Ende des umzutastenden Bereiches bereits mit normvideo abgeschlossen ist, da sonst ein häßliches Aufblitzen des Bildschirmes die Folge wäre. Diese notwendige Einfassung der Bildschirmtexte wird mit der lokalen Funktion norm zentral vorgenommen. Die Erzeugung und die Löschung einer Markierung werden durch die lokalen Prozeduren Markiere und DeMarkiere realisiert. Beide Prozeduren nutzen die für sie globale Variable hoehe, mit der jeweils spezifiziert wird, welche Position zu markieren oder zu demarkieren ist. Die Prozedur Markiere organisiert die Invers- und Intensiv-Tastung einer Menüzeile sowie die Ausgabe der zu dieser Zeile gehörenden Zusatzinformation. Diese Zusatzinformation, die zu jeder Menüposition rechts in der kleinen invers getasteten Tafel erscheint, ist in dieser, zur Funktion Menü lokalen Prozedur, als Konstantenfeld definiert. Jede Tafel kann maximal aus 3 Zeilen, die jeweils 15 Zeichen breit sind, bestehen. Durch die lokale Prozedur DeMarkiere wird diese Markierung einer Menüzeile wieder rückgängig gemacht, indem das Bildschirmsteuerzeichen ivitvideo am Anfang der Menüzeile wieder durch normvideo ersetzt wird. Das Löschen der alten Zusatztafel erfolgt durch einfa-

***Bild 7 Funktion Menü***

```
function Menue:char;
  const zpunkte='..............................................   ';
  var c:char;i,hoehe:integer;
  function norm(s:string80):string80;
    begin norm:=normvideo+s+normvideo end; (* Einfassen des strings *)
  procedure DeMarkiere;
    begin
      writexy(2,hoehe,normvideo);   (* Zeile wieder in Normaldarstellung *)
      for i:=hoehe to hoehe+2 do    (* alte Tafel wieder normal und leer *)
        writexy(54,i,norm('                ')));
    end;
  procedure Markiere;
    const tafel:array[10..20,1..3]of string[15]=
     ((' Beobachten von',                  (* Definition von 11 moeglichen  *)
       ' auftauchenden ',                  (* Tafeln zu den Menuepositionen *)
       '   Zeichen     '),
      (' Lokalisieren  ','    einer      ',' Bildschirmpos.'),
      (' Erfassen einer',' sechsstelligen','     Zahl      '),
      ('  Summieren von ','     fuenf     ','    Ziffern    '),
      ('','',''),('','',''),                (* Leere Tafeln fuer Leerposit.  *)
      ('','',''),('','',''),                (*  Reserve / keine Darstellung  *)
      (' 1 ist einfach ',' 3 ist gut     ',' 9 ist schwer  '),
      ('Zu viele Punkte','   ????????    ',' dann loeschen '),
      ('  Zurueck      ','       zum     ','        System '));
    begin
      writexy(2,hoehe,ivitvideo);   (*  Invers-Intensigve Menuezeile mit  *)
        for i:=hoehe to hoehe+2 do  (*  jeweils aktueller inverser Tafel  *)
          begin
            writexy(54,i,norm(tafel[hoehe,i-hoehe+1]));
            writexy(54,i,invsvideo)
    end   end;
begin
  clrscr;
  write(kursoroff);
  writeln(' K  K    OOO    N   N   ZZZZZ   ![!');
  writeln(' K K    O   O   NN  N      Z     ! ');
  writeln(' KK     O   O   N N N     Z      ! ');
  writeln(' K K    O   O   N  NN    Z       ! ');
  writeln(' K  K    OOO    N   N   ZZZZZ   ]!!');
  writexy(2,10,norm('.Welches.Zeichen....(Aufmerksamkeit).............A   '));
  writexy(2,11,norm('.Wo.war es ......(Aufmerksamkeit und Augenmasz)..B   '));
  writexy(2,12,norm('.Welche Zahl......(Erfassen.und.Merken)..........C   '));
  writexy(2,13,norm('.Rechnen..........(Erfassen.und.Rechnen).........D   '));
  for i:=14 to 17 do writexy(2,i,norm(zpunkte));
  writexy(2,18,norm('.Stellen Schwierigkeitsgrad......................S   '));
  writexy(2,19,norm('.Loeschen Punkte Stand...........................L   '));
  writexy(2,20,norm('.Ende............................................E   '));
  hoehe:=10;Markiere;

  repeat
    read(kbd,c);c:=upcase(c);
    DeMarkiere;                     (* Loeschen der alten Markierung *)
    case c of
      kursortief: if hoehe<20 then hoehe:=hoehe+1 else hoehe:=10;
      kursorhoch: if hoehe>10 then hoehe:=hoehe-1 else hoehe:=20;
      Enter     : case hoehe of
                    10:c:='A';11:c:='B';12:c:='C';13:c:='D';
                    18:c:='S';19:c:='L';20:c:='E' end
      end;
    Markiere;                       (* Markieren der neuen Position *)
  until c in ['A','B','C','D','S','L','E'];
  Menue:=c;
  write(kursoron)
end;
```

ches Überschreiben mit in normvideo eingefaßten Leerzeichenketten entsprechender Länge. Durch diese nicht zwingend notwendige Aufteilung von Unteraufgaben auf die beiden lokalen Prozeduren und die lokale Funktion ist der Programmtext der Funktion Menü sehr einfach und übersichtlich geworden.
Als erstes wird die Variable hoehe auf 10, das heißt auf die erste Menüposition gestellt und die Markierung durch Aufruf der Prozedur Markiere angewiesen. Die folgende Repeat-Schleife wird nun solange durchlaufen, bis

```
K   K    OOO    N   N   ZZZZZ   III
K K     O   O   NN  N      Z     I
KK      O   O   N N N     Z      I
K K     O   O   N  NN    Z       I
K   K    OOO    N   N   ZZZZZ   III

.Welches.Zeichen....(Aufmerksamkeit).............A
.Wo.war es ......(Aufmerksamkeit und Augenmasz)..B  Lokalisieren
.Welche Zahl......(Erfassen.und.Merken)..........C     einer
.Rechnen..........(Erfassen.und.Rechnen).........D  Bildschirmpos
..................................................
..................................................
..................................................
.Stellen Schwierigkeitsgrad......................S
.Loeschen Punkte-Stand...........................L
.Ende............................................E
```

*Bild 8 Bildschirmaussehen für das vorgestellte Auswahlmenü*

der Nutzer eine gültige Auswahl getroffen hat. Es wird jeweils ein Zeichen von der Tastatur abgefordert. Dieses Zeichen wird in einen Großbuchstaben umgewandelt und die alte Markierung auf dem Bildschirm mit Hilfe der Prozedur DeMarkiere gelöscht. Wurde eine der beiden möglichen Kursortasten betätigt, wird die Variable hoehe auf den neuen Wert eingestellt. Dabei wird eine Ringstruktur der Menüposition realisiert. Hat der Nutzer die Enter-Taste bedient, will er also die zuletzt markierte Option auswählen, wird das Zeichen durch den entsprechenden Auswahlbuchstaben ersetzt (Bild 7).
Die Verbindung zwischen dem Auswahlbalken und dem an das Hauptprogramm zu übergebenden Zeichen wird wieder über die Variable hoehe erreicht, über die nachfolgend auch wieder die Markierung der jeweils angewählten Menüzeile erfolgt. Entspricht nun dieses Zeichen einem möglichen Auswahlbuchstaben, kann die Funktion dieses Zeichen an das Hauptprogramm weitergeben. Wurde aber eine Kursortaste betätigt, wird die nächste Tastenbedienung des Nutzers erwartet. Das von der vorgestellten Menüfunktion erzeugte Bildschirmaussehen kann hier nur schematisch dargestellt werden. Dabei wird die eigentlich invers getastete Menüzeile und die Tafel mit den Zusatzinformationen nur durch Fettdruck gekennzeichnet (Bild 8). Der optische Reiz dieser Lösung besteht in der Dynamik des Menüs bei der Bedienung.
Durch die Einführung der Tafeln zu den einzelnen Menüpositionen bietet unser Menü dem Nutzer ausreichende Informationen zu den einzelnen auszuwählenden Aktionen. Reicht der Platz in den Tafeln nicht aus, kann die Festlegung der Tafelgröße in gewissen Grenzen, die durch die Bildschirmgröße gegeben sind, leicht verändert werden. Diese Ausbaustufe sollte auf kleineren Rechnern nicht mehr überschritten werden. Ein gewisser Service muß zwar realisiert werden, aber man sollte vermeiden, des Guten zu viel zu wollen. Die Anpassung der Funktion an weniger als 11 Menüpositionen ist einfach.

## Fenster-Hilfen

Beim Einsatz von Fenstertechniken ist es zum Beispiel möglich, in einem kleinen Bildschirmsegment bei sonst konstantem Bildschirmaussehen eine Vielzahl von Informationen vorbeirollen zu lassen. Dieses Rollen kann auf- oder abwärts erfolgen, je nach der speziellen Zweckmäßigkeit. Es ist auch denkbar, einzelne Fenster über den Bildschirm wandern zu lassen. Allerdings erfolgt bei dem hier vorgestellten Prinzip keine Rettung und Wiederherstellung von anderen überfahrenen Fensterinhalten.
Die vorzustellende Lösung wurde speziell für Compiler entwickelt, die ihrerseits die Window-, Insline- und Delline Befehle nicht realisieren und kann daher durch Nutzung dieser Befehle auf anderen Compilern wesentlich vereinfacht werden. Sie ist sofort auf allen Rechnern lauffähig, die als Prozessor den Z 80 (U 880) enthalten, und den Bildwiederholspeicher von 0F800H bis 0FF7FH aufbauen. Dies ist zum Beispiel bei jedem 8-Bit-PC oder -BC der Fall, sofern man darauf verzichtet, speicherplatzbeanspruchende Bildschirmsteuerzeichen zu verwenden. Um die Lauffähigkeit auf anderen Rechnern zu erreichen, ist aber nur die Änderung jener Komponenten der Lösung notwendig, die Inline-Maschinencode enthalten. Durch die Nutzung dieser Inline-Anweisungen konnte der Programm-Code dieser Lösung angemessen gering gehalten werden. Die Geschwindigkeit der Fensterprozeduren ist ebenfalls auf dieses Verfahren zurückzuführen. Die Prozeduren, die hier vorgestellt werden, können in zwei große Gruppen aufgegliedert werden. Die erste Gruppe ermöglicht die gewünschten Bewegungen auf dem Bildschirm. Dies sind vor allem die Prozeduren inslinedownin und inslineupin. Sie sind die Grundlage, auf der der Großteil der gesamten Lösung basiert. Aufbauend auf diese werden eine Reihe von Prozeduren angegeben, die den Programmierservice schaffen, der für die Beherrschung der Lösung notwendig ist.

*Bild 9 Ersatzprozeduren für Insline und Delline*

```
procedure inslinedown(i:integer);
  var z:integer;
  begin
    z:=(24-i)*80;
    inline($21/$2f/$4f/         (*LD HL,BS-Ende-80   *)
           $11/$7f/$f4/         (*LD DE,BS-Ende      *)
           $ed/$4b/z/           (*LD BC,(z)          *)
           $ed/$b8);            (*LDDR               *)
    gotoxy(1,i);write(#$16)     (*Loeschen i.Zeile   *)
  end;

procedure inslineup(i:integer);
  var z:integer;
  begin
    z:=(i-1)*80;
    inline($11/$00/$f8/       (*LD DE,BS-Anfang   *)
           $21/$50/$f8/       (*LD HL,Anfang 2.Zeile*)
           $ed/$4b/z/         (*LD BC,(z)         *)
           $ed/$b0);          (*LDIR              *)
    gotoxy(1,i);
    write(#$16);              (*Loeschen i.Zeile  *)
  end;

procedure inslinedownbis(i,u:integer);
  var w,z:integer;
  begin
    z:=(u-i)*80; u:=u*80-1; w:=u-80;
    inline($21/$00/$f8/   (*LD HL,BS-Anfang   *)
           $ED/$4b/u/     (*LD BC,(u)         *)
           $09/           (*ADD HL,BC         *)
           $EB/           (*EX DE,HL          *)
           $21/$00/$f8/   (*LD HL,BS-Anfang   *)
           $ed/$4b/w/     (*LD BC,(w)         *)
           $09/           (*ADD HL,BC         *)
           $ed/$4b/z/     (*LD BC,(z)         *)
           $ed/$b8);      (*LDDR              *)
    gotoxy(1,i);write(#$16)
  end;
```

## Grundprozeduren

Als erstes werden in Bild 9 zwei Prozeduren vorgestellt, die als Ersatz für die nicht zur Verfügung stehenden Standardprozeduren Insline und Delline erstellt wurden. Sie sind hier angeführt, um ein schnelleres Verständnis der folgenden physischen Grundprozeduren zu erreichen. In beiden Prozeduren wird auf der durch den Parameter i spezifizierten Zeile des Bildschirmes eine Leerzeile eingefügt. Von dieser Zeile an wird der ursprüngliche Bildschirminhalt verschoben, wobei die letzte bzw. die erste Bildschirmzeile verloren geht. Dieses Wegschieben erfolgt bei der Prozedur inslinedown nach unten und bei der Prozedur inslineup nach oben. Am Anfang beider Prozeduren wird zuerst die Errechnung der Anzahl der zu verschiebenden Bildschirmzeichen vorgenommen, die leicht nachzuvollziehen ist. Da das Verschieben der Zeilen direkt im Bildwiederholspeicher mit Hilfe der Befehle LDDR und LDIR erfolgt, sind die Rechenzeiten dieser Prozeduren minimiert. Das Löschen der eingefügten Bildschirmzeilen wird mit Hilfe eines Bildschirmsteuerzeichens angewiesen, welches eine komplette Zeile löscht und den Kursor an deren Anfang setzt. Durch dieses Vorgehen erhält man eine sehr gute Bilddynamik. Nach der Abarbeitung einer der beiden Prozeduren steht der Kursor am Anfang der eingefügten Zeile.

*wird fortgesetzt*

# Turbo-Pascal-Praxis

Teil 2

**Manfred Zander, Dresden**

*Im Teil 1 unseres Kurses (s. MP 6/89, S. 175) wurden die Programmierung einer positionierten, absturzfesten Eingabe und die Auswahl mit Menüs dargestellt sowie die Arbeit mit Fenstern eingeleitet. Wir endeten mit dem Bild 9 und dem Vorstellen der Grundprozeduren. Der Teil 2 setzt nun hier fort und behandelt den Komplex Fenstertechnik.*

Mit der Prozedur inslinedown (vgl. Bild 9 im Teil 1) ist es nun möglich, unter einem Tabellenkopf, der fest auf dem Bildschirm steht, immer wieder neue Zeilen zu erfassen, und bereits erfaßte auch unten wegrollen zu lassen. Hier ein kurzer Programmauszug, der dies zeigt.

```
writexy(1,10,'Name            Vorname');
writexy(1,11,'--------------------------------');
repeat
  inslinedown(12);
  read(namen[i]);gotoxy(20,12);
  read(Vornamen[i];i:=i+1
until namen [i−1]='';
```

Besteht der Wunsch, das Wegrollen der Zeilen nur bis zu einer bestimmten Zeile zuzulassen, das heißt sowohl einen Tabellenkopf als auch einen Tabellenfuß fest auf dem Bildschirm zu belassen, kann die Prozedur inslinedownbis (Bild 9) genutzt werden. Sie schränkt den Bereich vertikal ein, in dem das Wegrollen der Zeilen erfolgen soll. Zuerst errechnet sie die Anzahl der zu verschiebenden Bildschirmzeichen, die in der Variablen z gespeichert wird. Für die Variablen u und w werden Offsetwerte bestimmt, gerechnet ab der Adresse des Bildschirmanfangs. Der Offsetwert u gibt dabei die Adresse für das erste zu verschiebende Zeichen, und der Offsetwert w die neue Addresse für dieses Zeichen an. Die Errechnung der wirklichen Adressen erfolgt erst in der inline-Anweisung, in der wieder ein zyklischer Ladebefehl des Prozessors genutzt wird.
Die beiden Parameter der Prozedur inslinedownbis spezifizieren in ihrer Reihenfolge die erste Zeile, die wegrollen soll, also die Position der einzufügenden Leerzeile und die letzte wegzurollende Zeile, das heißt die Zeile, die aus dem Bereich herausgerollt und damit vom Bildschirm gelöscht werden soll. Damit ist dies die erste Prozedur, die eine Art Fenster auf dem Bildschirm behandelt. Der Aufruf:

Inslinedownbis(10,13);

würde die folgende Zeilenbewegung hervorrufen:

10. Zeile auf die 11. Zeile
11. Zeile auf die 12. Zeile
12. Zeile auf die 13. Zeile

Die 10. Zeile ist danach leer, und der Inhalt der ursprünglichen 13. Zeile ist nicht mehr auf dem Bildschirm vorhanden. Wir haben also eine vertikale Fensterbegrenzung erreicht.

Nun gilt es auch noch, eine horizontale Eingrenzung zu erreichen, wie es ja für ein Fenster typisch ist. Als erstes wollen wir die Prozedur inslinedownin vorstellen (Bild 10). In ihr wird zuerst wieder die Länge der Teilzeilen bestimmt, die auf dem Bildschirm nach unten verschoben werden sollen. Da diese Teilzeilen nun nicht mehr lückenlos im Hauptspeicher hintereinander stehen, muß die Berechnung der Offsetadressen für jede Zeile, die verschoben werden soll, einzeln erfolgen. Mit der Inline-Anweisung wird nun auch jeweils nur eine Teilzeile auf ihre neue Position gebracht.
Das abschließende Löschen der einzufügenden Zeile am oberen Rand des zu behandelnden Fensters kann nun nicht mehr durch die Ausgabe eines Bildschirmsteuerzeichens erfolgen. Wir sind gezwungen, jedes einzelne Zeichen dieser Fensterzeile mit einem Leerzeichen zu überschreiben und den Kursor danach an ihren Anfang zu positionieren.
Die Prozedur inslinedownin benötigt nun bereits vier Parameter, die den zu verschiebenden Bereich eingrenzen. X1 und Y1 definieren den oberen linken Eckpunkt des Bereiches, die Parameter X2 und Y2 den unteren Eckpunkt des zu rollenden Bereiches auf dem Bildschirm. Mit dieser Prozedur sind wir jetzt in der Lage, die oberste Zeile des angegebenen Fensters mit einer Leerzeile zu füllen und alle anderen im Fenster enthaltenen Zeilen nach unten wegrollen zu lassen. Dabei bleibt der komplette Bildschirm, der nicht innerhalb der beiden Eckpunkte liegt, unverändert. Ein typischer Aufruf der Prozedur wäre beispielsweise

```
procedure inslinedownin(x1,y1,x2,y2:integer);
  var i,o,u,z:integer;
  begin
    z:=x2-x1+1;
    for i:=y2 downto y1+1 do begin
      u:=(i-1)*80+x2-1; o:=u-80;
      inline($21/$00/$f8/  (*LD HL,BS-Anfang   *)
             $ED/$4b/u/    (*LD BC,(u)         *)
             $09/          (*ADD HL,BC         *)
             $EB/          (*EX DE,HL          *)
             $21/$00/$f8/  (*LD HL,BS-Anfang   *)
             $ed/$4b/o/    (*LD BC,(o)         *)
             $09/          (*ADD HL,BC         *)
             $ed/$4b/z/    (*LD BC,(z)         *)
             $ed/$b8);end; (*LDDR              *)
    gotoxy(x1,y1);
    for i:=x1 to x2 do write(' ');
    gotoxy(x1,y1)
  end;


procedure inslineupin(x1,y1,x2,y2:integer);
  var i,o,u,z:integer;
  begin
    z:=x2-x1+1;
    for i:=y1+1 to y2 do begin
      u:=(i-1)*80+x2-1; o:=u-80;
      inline($21/$00/$f8/   (*LD HL,BS-Anfang   *)
             $ED/$4b/o/     (*LD BC,(o)         *)
             $09/           (*ADD HL,BC         *)
             $EB/           (*EX DE,HL          *)
             $21/$00/$f8/   (*LD HL,BS-Anfang   *)
             $ed/$4b/u/     (*LD BC,(u)         *)
             $09/           (*ADD HL,BC         *)
             $ed/$4b/z/     (*LD BC,(z)         *)
             $ed/$b8);end;  (*LDDR              *)
    gotoxy(x1,y2);
    for i:=x1 to x2 do write(' ');
    gotoxy(x1,y2)
  end;
```

*Bild 10 Grundprozeduren der Fenstertechnik*

**Inslinedownin(30,10,70,15);**

Durch geeignete Wahl der Eckpunkt-Parameter kann mit dieser Prozedur auch die Funktion der bereits vorgestellten Prozeduren inslinedown und inslinedownbis nachgestaltet werden. Natürlich wollen wir auch in der Lage sein, innerhalb eines Fensters auf der letzten Zeilenposition eine Leerzeile einzufügen und dabei alle anderen Zeilen innerhalb des Fensters nach oben wegrollen zu lassen. Dies wird mit der folgenden Prozedur inslineupin erreicht. Da ihre innere Struktur fast völlig der Prozedur inslinedownin entspricht, muß ihre Funktion nicht näher erläutert werden.

### Arbeiten mit den Fenstern

Mit den beiden Prozeduren inslinedownin und inslineupin haben wir jetzt die notwendigen Hilfsmittel, um auf dem Bildschirm mit Fenstern operieren zu können. Die Benutzung dieser Prozeduren für mehere Fenster ist aber recht mühsam, da ihnen stets die Eckpunktkoordinaten des zu behandelnden Fensters übergeben werden müssen. Es ergibt sich also die Notwendigkeit, auch einige logische Hilfsmittel zur Nutzung der Fenster zu entwerfen, die uns die Verwaltung der konkreten Koordinaten abnehmen. Dies wird vor allem dann notwendig, wenn wir die Fenster samt Inhalt über den Bildschirm verschieben wollen, da sich die aktuellen Eckpunkte nach jeder Verschiebeoperation natürlich ändern. Zur Verwaltung der Fensterkoordinaten nutzen wir ein Feld, in dem die Informationen gesammelt werden, die die jeweils deklarierten Fenster kennzeichnen (Bild 11).
In dem definierten Array wins können nun die Daten von fünf Fenstern gespeichert werden. Dabei ist jedes Fenster durch vier Integerwerte definiert, die die Eckpunkt-Koordinaten der Fenster repräsentierten, sowie durch ein Zeichen, in dem wir den ASCII-Wert speichern wollen, mit dem das Fenster bei Bedarf umrahmt werden soll. Zum Füllen dieser Felder nutzen wir die folgende Prozedur windowdef. Durch diese Vereinbarungen zur Verwaltung der Fensterwerte können wir nun die Bedienung der Prozeduren inslinedownin und inslineupin wesentlich vereinfachen. Die beiden Prozeduren InWindowDown und InWindowUp übernehmen die korrekte Füllung der Parameterlisten der Prozeduren inslinedownin und inslineupin mit den Koordinaten, die für das zu behandelnde Fenster im Array wins gespeichert sind.
Wurden mit Hilfe der Prozedur windowdef die Koordinaten der Fenster einmalig festgelegt, braucht sich der Programmierer also im weiteren nicht mehr um die konkreten Koordinaten der einzelnen Fenster zu kümmern. Durch die Anwendung der neuen Prozeduren InWindowDown und InWindowUp genügt es völlig, die Nummer des zu behandelnden Fensters anzugeben. Diesen Service wollen wir auch für alle weiteren Fenster-Behandlungen erreichen. Als nächstes wollen wir

```
const FensterAnzahl = 5;
var   wins : array[1..FensterAnzahl] of
             record
               x1,y1,x2,y2:integer;
               c:char
             end;


procedure windowdef(i,x1,y1,x2,y2:integer;c:char);
  begin
    wins[i].x1:=x1;      wins[i].x2:=x2;
    wins[i].y1:=y1;      wins[i].y2:=y2;
    wins[i].c:=c
  end;


procedure InWindowDown(i:integer);
  begin
    inslinedownin(wins[i].x1,wins[i].y1,wins[i].x2,wins[i].y2)
  end;


procedure InWindowUp(i:integer);
  begin
    inslineupin(wins[i].x1,wins[i].y1,wins[i].x2,wins[i].y2)
  end;
```

*Bild 11 Logische Bedienung der Fenster*

```
procedure Rahmen(x1,y1,x2,y2:integer;c:char);
  var i:integer;
  begin
    write(kursoroff);gotoxy(x1,y1);
    for i:=x1 to x2 do if c='|' then write('-') else write(c);
    for i:=y1+1 to y2-1 do
      begin
        gotoxy(x1,i);write(c); gotoxy(x2,i);write(c)
      end;
    gotoxy(x1,y2);
    for i:=x1 to x2 do if c='|' then write('-') else write(c);
    write(kursoron)
  end;


procedure WindowWrite(i:integer);
  begin
    Rahmen(wins[i].x1-1,wins[i].y1-1,wins[i].x2+1,wins[i].y2+1,wins[i].c)
  end;
```

*Bild 12 Umrahmen von Fenstern*

```
procedure WriteDown(i:integer;str:string80);
  var l:integer;
  begin
    InWindowDown(i);
    l:=wins[i].x2-wins[i].x1+1;
    if length(str)> l then str:=copy(str,1,l);
    write(str)
  end;


procedure WriteUp(i:integer;str:string80);
  var l:integer;
  begin
    InWindowUp(i);
    l:=wins[i].x2-wins[i].x1+1;
    if length(str)> l then str:=copy(str,1,l);
    write(str)
  end;


function holeRin(i:integer;str:string80):real;
  var Io:integer;
      s:string80;
      r:real;
  begin
    repeat
      writeUp(i,str+'=');          (* Ausgabe der Eingabeaufforderung *)
      read(s);                     (* Einlesen eines Strings          *)
      val(s,r,Io)                  (* Umwandlung des Strings in real  *)
    until Io=0;                    (* bis Umwandlung Io ist           *)
    holeRin:=r
  end;


procedure ReadWriteInWindow(i:integer;var str:string80);
  var l,d:integer; c:char;
  begin
    inwindowdown(i); str:='';d:=0;
    l:=wins[i].x2-wins[i].x1+1;
    repeat
      read(kbd,c);
      if c<>#13 then write(c);
      str:=str+c;d:=d+1;
      if c = #8 then
        begin
          str:=copy(str,1,length(str)-2);
          write(#$20,#8);d:=d-2;
        end;
      if c = #127 then d:=d-2
    until (c = #13) or (l=d)
  end;
```

*Bild 14 Schreiben und Lesen in Fenstern*

unsere Fenster umrahmen. Dazu schreiben wir zuerst die Prozedur Rahmen (Bild 12), die die konkreten Koordinaten benötigt, und dann die Prozedur WindowWrite, die die erste leichter bedienbar macht. Die Prozedur Rahmen realisiert das Umrahmen eines Bildschirmbereiches mit dem angegebenen ASCII-Zeichen. Eine Sonderbehandlung erfährt das Zeichen |. Wird es angegeben, so werden die waagerechten Begrenzungen nicht mit dem ASCII-Zeichen selbst, sondern mit einem Minus-Zeichen aufgebaut. Die Prozedur WindowWrite realisiert in bewährter Weise die Bereitstellung der konkreten Koordinaten und des ASCII-Zeichens, welches für den Rahmen verwendet werden soll. Zu ihrer Nutzung ist wiederum lediglich die Angabe der Fenster-Nummer notwendig. Hier nun wollen wir ein kurzes Beispiel für die Nutzung und die Wirkung der bisherigen Prozeduren angeben:

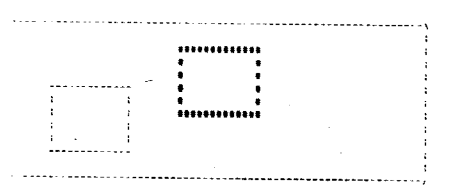


*Bild 13 Bildschirm mit umrahmten Fenstern*

```
...
Windowdef(1,10,3,20,6,'|');
Windowdef(2,30,6,40,9,'#');
WindowWrite(1);
WindowWrite(2);
...
```

Mit Hilfe dieser vier Anweisungen wird das in Bild 13 veranschaulichte Bildschirmaussehen realisiert.

In Bild 14 folgen zwei Prozeduren, mit denen wir in diese nun definierten und auf dem Bildschirm gekennzeichneten Fenster schreiben können. Es sind deshalb zwei Prozeduren vorgesehen, da wir ja das Rollen in den Fenstern sowohl aufwärts als auch abwärts ermöglichen wollen. Die Prozedur WriteDown, der die Nummer des betreffenden Fensters und der auszugebende Text übergeben wird, rollt zuerst den kompletten Fensterinhalt um eine Zeile nach unten. Der neu in das Fenster zu schreibende Text wird dann in die geschaffene Leerzeile am oberen Rand des Fensters geschrieben. Die Prozedur WriteUp dagegen schiebt den Fensterinhalt nach oben und schreibt auf der untersten Fensterzeile.

In beiden Prozeduren wird garantiert, daß zu langer Text nicht über das jeweilige Fenster hinaus geschrieben wird, sie sorgen selbständig für eine neue Zeile im Fenster, auf der der Text dann geschrieben wird. Bild 15 zeigt das erzeugte Bildschirmaussehen. Selbstverständlich müssen die genutzten Fenster 1 und 2 vor diesen Anweisungen erst mit der Prozedur windowdef deklariert und mit der Prozedur WindowWrite gezeichnet worden sein.

```
...
writeDown(1,'Erster');
writeUp(2,'Zweiter');
writeDown(1,'Dritter');
writeUp(2,'Vierter');
writeUp(2,'Zu langer Text für das Fenster');
...
```

So wie wir mit den Prozeduren WriteDown und WriteUp in die Fenster geschrieben haben, ist es natürlich auch möglich, in den Fenstern positioniert zu lesen. Allerdings ist es hierbei kaum möglich, Prozeduren zu entwik-

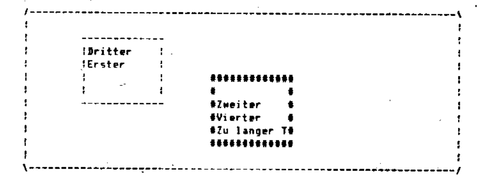



*Bild 15 Der Rahmen begrenzt das Fenster, auch wenn der Text zu lang ist.*

keln, die allen möglichen Ansprüchen gleichermaßen genügen. Hier soll nur eine Variante angeboten werden. Jeder Programmierer kann sich dann selbst schreiben, was für das jeweils von ihm zu lösende Problem passend ist. Die folgende Funktion holeRin bewirkt die Ausgabe eines Nutzertextes auf die im Fenster eingefügte Leerzeile und dahinter das Einlesen einer Real-Zahl. Das Abfangen von Fehleingaben wurde selbstverständlich ebenfalls gleich in der Funktion selbst realisiert.
Für die Nutzung der Funktion holeRin wollen wir uns wieder ein kleines Anwendungsbeispiel ansehen:

```
...
writeUp(1,'Maße fuer');
writeUp(1,'Kreiszyl.');
Ra:=holeRin(1,'Radius');
Ho:=holeRin(1,'Höhe');
...
```

Wenn wir annehmen, daß sich der Bediener einmal vertippt hat, so ergibt sich das in Bild 16 dargestellte Bildschirmaussehen. Dabei soll nicht verschwiegen werden, daß die angegebene Funktion keinen Schutz davor bietet, daß der Bediener aus dem Fenster hinausschreibt. Einen Schutz vor dem Überschreiben des Fensterbereiches bei der Eingabe von Daten durch den Nutzer gewährleistet zum Beispiel die Prozedur ReadWriteInWindow (Bild 14). Die erfordert aber eine etwas vom Gewohnten abweichende Eingabeführung und ist somit nicht überall anwendbar.
Nun fehlen uns noch einfache Hilfsmittel, die den Inhalt eines Fensters oder ein komplettes Fenster mit Rahmen vom Bildschirm löschen. Um den Aufwand hierfür so gering wie möglich zu belassen, nutzen wir wieder schon vorhandene Prozeduren. Die Prozedur windowclear (Bild 17) schiebt den betreffenden Fensterinhalt nach unten hinaus, indem sie die dafür erforderliche Anzahl von Leerzeilen in das Fenster einfügt. Die Nutzung der Prozedur InWindowDown für diese Aufgabe ist willkürlich gewählt. Mit der Prozedur InWindowUp könnte man den Fensterinhalt auch nach oben wegschieben. Die in Bild 17 angegebene Prozedur killwindow ist nicht zwingend notwendig. Ihre Funktion in der hier angegebenen Form basiert komplett auf bereits vorgestellten Komponenten der Fenstertechnik. Zuerst erfolgt die logische Vergrößerung des Fensters, welches vom Bildschirm gelöscht werden soll. Das neu definierte Fenster enthält nun auch den Rahmen des ursprünglich definierten. Somit entspricht das Löschen des neuen Fensterinhalts dem Löschen des alten Fensters mit Rahmen.

### Verschieben von Fenstern

Wir haben uns nun alle Prozeduren geschaffen, um mit den fest deklarierten Fenstern zu arbeiten. Es fehlen nur noch Hilfsmittel, um bereits deklarierte und gezeichnete Fenster logisch und auf dem Bildschirm zu verschieben. Diese vier Prozeduren sind alle so geschrieben, daß die Fenster, an den Rand des Bildschirmes gefahren, immer kleiner werden. Konflikte treten erst dann auf, wenn die Fenster nur noch eine Zeile hoch oder eine Spalte breit sind. Um diese Konflikte zu vermeiden, sind äußere Maßnahmen erforderlich. Das Wandern der Fenster auf dem Bildschirm wird mit den Prozeduren in den Bildern 18 und 19 angewiesen. Dabei wurde, wenn möglich, wieder auf bereits vorhandenes zurückgegriffen. Das Wirkprinzip der Prozeduren windowtief und windowhoch (Bild 18) ähnelt dem der Prozedur kollwindow. Es wird dabei davon ausgegangen, daß das Verschieben eines kompletten Fensters nach unten oder nach oben genau dem Verschieben eines Fensterinhaltes mit korrigierten Koordinaten äquivalent ist. Nach dem Verschieben des Fensters auf dem Bildschirm wird dann die Berichtigung der neuen gültigen Eckpunkt-Werte im array wins vorgenommen.
Für das Wandern der Fenster nach rechts bzw. nach links war es aber notwendig, wieder **InLine-Maschinen-Code** zu nutzen, da ja keine Prozeduren für ein horizontales Verschieben von Fensterinhalten vorgesehen sind. Der innere Aufbau der Verschiebe-Prozeduren windowrechts und windowlinks (Bild 19) entspricht vom Prinzip her wieder

```
/--------------------------------------------------------------\
!                                                              !
!        -------------                                         !
!        !Kreiszyl.  !                                         !
!        !Radius= 5,6!                                         !
!        !Radius= 5.6!      #############                      !
!        !Hoehe = 3  !      #           #                      !
!        -------------      #Zweiter    #                      !
!                           #Vierter    #                      !
!                           #Zu langer T#                      !
!                           #############                      !
!                                                              !
\--------------------------------------------------------------/
```

*Bild 16 Bildschirm nach der Eingabe in ein Fenster*

```
procedure windowclear(i:integer);
  var z:integer;
  begin
    write(korsoroff);
    for z:=0 to wins[i].y2-wins[i].y1 do
      InWindowDown(i);
    write(kursoron)
  end;


procedure killwindow(i:integer);
  begin
    windowdef(i,wins[i].x1-1,wins[i].y1-1,wins[i].x2+1,wins[i].y2+1,' ');
    windowclear(i)
  end;
```

*Bild 17 Lösch-Prozeduren*

```
procedure windowtief(i:integer):
  begin
    if wins[i].y2=23 then wins[i].y2:=22;
    inslinedownin(wins[i].x1-1,wins[i].y1-1,wins[i].x2+1,wins[i].y2+2);
    windowdef(i,wins[i].x1,wins[i].y1+1,wins[i].x2,wins[i].y2+1,wins[i].c);
    if wins[i].y2=23 then windowwrite(i)
  end;


procedure windowhoch(i:integer);
  begin
    if wins[i].y1=2 then wins[i].y1:=3;
    inslineupin(wins[i].x1-1,wins[i].y1-2,wins[i].x2+1,wins[i].y2+1);
    windowdef(i,wins[i].x1,wins[i].y1-1,wins[i].x2,wins[i].y2-1,wins[i].c);
    if wins[i].y1=2 then windowwrite(i)
  end;
```

*Bild 18 Vertikales Verschieben von Fenstern*

*Bild 19 Horizontales Verschieben von Fenstern*

```
procedure windowrechts(i:integer);
  var v,h,z,j:integer;
  begin
    if wins[i].x2=79 then wins[i].x2:=78;
    z:=wins[i].x2-wins[i].x1+3;
    for j:=wins[i].y1-1 to wins[i].y2+1 do
      begin
        v:=(j-1)*80+wins[i].x2;h:=v+1;
        inline($21/$00/$f8/   (* LD HL,BS-Anfang *)
               $ed/$4b/h/     (* LD BC,(h)       *)
               $09/           (* ADD HL,BC       *)
               $eb/           (* EX DE,HL        *)
               $21/$00/$f8/   (* LD HL,BS-Anfang *)
               $ed/$4b/v/     (* LD BC,(v)       *)
               $09/           (* ADD HL,BC       *)
               $ed/$4b/z/     (* LD BC,(z)       *)
               $ed/$b8);      (* LDDR            *)
      end;
    for j:=wins[i].y1-1 to wins[i].y2+1 do
      begin
        gotoxy(wins[i].x1-1,j);write(' ')
      end;
    wins[i].x1:=wins[i].x1+1;wins[i].x2:=wins[i].x2+1;
    if wins[i].x2=79 then windowwrite(i)
  end;

procedure windowlinks(i:integer);
  var v,h,z,j:integer;
  begin
    if wins[i].x1=2 then wins[i].x1:=3;
    z:=wins[i].x2-wins[i].x1+3;
    for j:=wins[i].y1-1 to wins[i].y2+1 do
      begin
        v:=(j-1)*80+wins[i].x1-3;h:=v+1;
        inline($21/$00/$f8/   (* LD HL,BS-Anfang *)
               $ed/$4b/v/     (* LD BC,(v)       *)
               $09/           (* ADD HL,BC       *)
               $eb/           (* EX DE,HL        *)
               $21/$00/$f8/   (* LD HL,BS-Anfang *)
               $ed/$4b/h/     (* LD BC,(h)       *)
               $09/           (* ADD HL,BC       *)
               $ed/$4b/z/     (* LD BC,(z)       *)
               $ed/$b0);      (* LDIR            *)
      end;
    for j:=wins[i].y1-1 to wins[i].y2+1 do
      begin
        gotoxy(wins[i].x2+1,j);write(' ')
      end:
    wins[i].x1:=wins[i].x1-1;wins[i].x2:=wins[i].x2-1 ;
    if wins[i].x1=2 tHen windowwrite(i)
  end;
```

den am Anfang vorgestellten Prozeduren inslinedwonin und inslineupin. Schließlich ist das horizontale Verschieben ja auch wieder nichts weiter als ein Einfügen einer Leerspalte mit Verschiebung des restlichen Fensterinhalts. Auch diese beiden Prozeduren bewirken neben dem Verschieben des kompletten Fensters auf dem Bildschirm wieder eine Korrektur der Eintragungen im Array wins.

### Beispiele für Fenstertechnik

Insgesamt wurden zwanzig Prozeduren bzw. Funktionen der Fenstertechnik entwickelt. Will man nur die Prozeduren inslinedown, inslineup oder inslinedownbis nutzen, so genügt es völlig, nur diese in ein Programm aufzunehmen. Anders sieht es damit aus, wenn man horizontal und vertikal begrenzte Fenster nutzen will. Dann sollten diese Lösungen auch das Array wins und die Prozedur windowdef enthalten. Da die meisten Prozeduren auf untergeordnete zurückgreifen, muß jeweils ermittelt werden, wo das Ende der Verschachtelung ist. Will man beispielsweise die Prozedur WriteDown nutzen, so muß das Programm auch die Prozeduren InWindowDown und inslinedownin beinhalten. Die für ein spezielles Programm benötigten Komponenten sollten dann in einem Include-File zusammengefaßt werden. In diesem kann man dann die teilweise sehr reichhaltigen Kommentare des Quelltextes löschen und die optische Strukturierung der Prozeduren auf ein Minimum reduzieren. Diese beiden Maßnahmen dienen der Verringerung von Quelltextgröße und Kompilierzeit.

Als Beispiel für die Anwendung der vorgestellten Fenstertechnik folgt jetzt ein Auszug aus einem Programm, in dem die Masse von geometrisch bestimmten Körpern berechnet wird (Bild 20). Da es notwendig ist, die jeweilige Dichte des entsprechenden Werkstoffes einzugeben, werden im Fenster mit der Nummer 1 unter anderem verschiedene Materialien angeboten, die mit jeweils einem Kennbuchstaben ausgewählt werden können. Die Anzahl der angebotenen Materialien kann aber die Zeilenzahl des Bildschirms überschreiten; es wurde eine Fenster-Lösung erforderlich. Die Funktion dieser Auswahlmenü-Lösung ist folgendermaßen:

Die ersten 19 Materialien werden in dem Fenster mit ihren jeweiligen Kennungen angeboten. Ist das vom Nutzer benötigte Material nicht dabei, erhält er durch Betätigung der Entertaste die nächsten 19 angeboten. Die Auswahl eines Materials erfolgt durch die Eingabe seines Kennbuchstabens.

Weiterhin verdient die Funktion lesen im Programmbeispiel Beachtung. Sie entspricht fast völlig der vorgeschlagenen Funktion holeRin unserer Fenstertechnik. Dadurch, daß in ihr das Fenster, in dem gelesen werden soll, fest eingestellt ist, wird zentral geregelt, in welchem Bereich des Bildschirmes die Eingabe von Zahlen erfolgen soll. Durch diese Festlegungen ist es gelungen, auf dem Bildschirm thematisch getrennte Bereiche einfach festzulegen und zu bedienen.

Zur Veranschaulichung des Programms ist in Bild 21 das Aussehen des Bildschirmes dargestellt. Die Pfeile geben dabei die Bewegungsrichtung der Fensterinhalte beim Rollen an. Es folgt noch ein weiteres Beispiel, dieses Mal vor allem für die Anwendung der Prozeduren zum Verschieben der Fenster auf dem Bildschirm (Bild 22). Das Demonstrationsbeispiel realisiert das dreimalige Umkreisen des Bildschirm-Mittelpunktes mit einem Fenster entgegen dem Uhrzeigersinn. Nach der Aufwärtsbewegung des Fensters wird der optische Effekt erzeugt, daß der Inhalt des Fensters auf Grund der Trägheit noch weiter nach oben steigt, und dann erst im Fenster herunterfällt.

```
program Massenberechnung;
type string80=string[80];
     suchtyp = array[1..40]of record
                 txt:string[40];kenn:char;wert:real end;
var wahl:char;
    Dichten:suchtyp;
    masse,volumen,dichte:real;
    wins :array[1..5] of record x1,y1,x2,y2:byte;c:char end;
{$I windows.inc }
procedure Bild;
  begin
    clrscr;
    windowdef(1,2,2,39,21,'!');windowwrite(1);   (* Menue  Fenster          *)
    windowdef(2,41,2,79,14,'!');windowwrite(2);  (* Kontrol/Eingabe Fenster*)
    windowdef(3,41,16,78,23,'!');windowwrite(3); (* Ergebnis  Fenster       *)
    gotoxy(1,23);write('Wahl Buchstabe ! ET fuer weiter');
    writeDown(3,'Faktor:              !Masse:           ');
    writeDown(3,'Aufrechnungs-        !Total-');
    writeDown(3,'---------------------!-------------------');
    writeDown(3,'Volum:               !Volum:');
    writeDown(3,'Letztes Einzel-      !Total-');
    writeDown(3,'---------------------!-------------------');
    writeDown(3,'Dicht:               !');
    writeDown(3,'letzte Einzel-       !');
  end;
function lesen(textt:string80):real;
  var zeil:string80;i:integer;r:real;
  begin
    repeat
      writeUp(2,textt+' = ');     (* Mit dieser function wird  *)
      read(zeil);val(zeil,r,i);   (* realisiert, dasz alle     *)
    until i =0;                   (* Zahleneingaben im Fenster *)
    lesen:=r;                     (* Nr. 2 erfolgen            *)
  end;
procedure kennhol(M:Suchtyp;var kenn:char;var dichte:real);
  var i:integer;
  begin
    windowclear(1);                             (* Da das Fenster Nr.1 *)
    i:=1;Dichte:=0.0;                           (* auch fuer andere    *)
    repeat                                      (* Menues genutzt wird *)
      repeat                                    (* wird das Menue als  *)
        if m[i].txt='' then i:=1;               (* Parameter uebergeben*)
        writeDown(1,m[i].txt+'    '+m[i].kenn);(* Es werden 19 neue   *)
        i:=i+1                                  (* Zeilen ins Menue    *)
      until i=19;                               (* Fenster geschoben   *)
      read(kbd,kenn);(* Nutzerabfrage *)
    until kenn<>#13; (* Neue Menuezeilen bei Betaetigung der ET-Taste *)
    i:=0;
    repeat  i:=i+1                               (* Suchen ausgewaehlter  *)
      until (m[i].kenn=kenn)or(m[i].txt='');     (* MenuePosition im Feld *)
    if m[i].txt<>'' then
      begin                    (* Wenn ausgewaehlte Position existiert:  *)
        writeUp(2,m[i].txt);   (* Auswahl ins Kontroll-Fenster schreiben *)
        dichte:=m[i].wert      (* Ausgewaehlten Wert uebernehmen         *)
  end end;
begin    (* Hauptprogramm *)
  Bild;
  ...........(* Fuellen des Feldes Dichten und Ermittelung des Volumens *)
  kennhol(Dichten,wahl,dichte);
  if (upcase(wahl)='F')or(dichte=0.0) then dichte:=lesen('Direkt-Wert');
  gotoxy(49,17);write(dichte:10:4);
  masse:=Volumen*dichte;
  gotoxy(67,23);write(masse:12:3)
end.
```

*Bild 20 Programm Masse*

```
------------------------------------------------------------------------------
! Silber         S                   !                                       !
! Quecksilber    q          !        !                     /^\               !
! Quarzglas      Q          !        !                      !                !
! Porzellan      p          !        !                      !                !
! Platin         P          !        !                      !                !
! Nickel         n         \v/       !                      !                !
! Natrium        N                   !                                       !
! Marmor         M                   !Radius = 1                             !
! Kobalt         k                   !V = 12                                 !
! Konstantan     K                   !Direkt-Wert = 12                       !
! Gold           G                   !Aluminium                              !
! Kork           c                   !Hardcopy                               !
! Chrom          C                   --------------------------------------
! Barium         b                   !letzte Einzel-     !                  !
! Blei           B                   !Dicht:     6.69000 !                  !
! Antimon        a                   !-------------------!------------------!
! Aluminium      A                   !Letztes Einzel-    !Total-            !
! Direkt-Wert    F/f                 !Volum:   12.000000 !Volum:   48.566371!
!                                    !-------------------!------------------!
-------------------------------------Aufrechnungs-       !Total-            !
Wahl Buchstabe ! ET fuer weiter      !Faktor: 3          !Masse:   84.069019!
                                     --------------------------------------
```

*Bild 21 Bildschirmdarstellung des Masse-Programms*

```
program FensterLauf;             (* Beispiel fuer Fensterverschieben *)
type string80 = string[80];
var i,j:integer;
{$I windows.ovl}
begin
  write(kursoroff);
  windowdef(1,10,5,20,10,'!');
  windowwrite(1);
  for j:=1 to 3 do
    begin
      for i:=1 to 10 do windowtief(1);   (* Fallen des Fensters           *)
      for i:=1 to 30 do windowrechts(1);(* Fenster wandert nach rechts   *)
      for i:=1 to 10 do windowhoch(1);   (* Steigen des Fensters          *)
      for i:=1 to 4 do inwindowup(1);    (* Weitersteigen  Fensterinhalt *)
      for i:=1 to 4 do inwindowdown(1);  (* Herunterfallen Fensterinhalt *)
      for i:=1 to 30 do windowlinks(1);  (* Fenster wandert nach links   *)
    end;
  killwindow(1);
  write(kursoron)
end.
```

*Bild 22 Programm FensterLauf*

*wird fortgesetzt*

# Turbo-Pascal-Praxis

Teil 3

**Manfred Zander, Dresden**

*Wir haben uns im Teil 2 unseres Kurses in MP 8/89 der Programmierung der Fenstertechnik für den PC 1715 und dazu kompatibler Rechner gewidmet. Auf den folgenden 4 Seiten sollen neben der Arbeit mit Menü- und Hilfsausschriften sowohl Hardcopy-Funktionen als auch Prozeduren zur Anzeige des Disketten-Directories dargeboten werden.*

## Bildschirm-Darstellungen

In vielen Programmen, die für Nutzer gedacht sind, die noch nie etwas mit Rechentechnik zu tun hatten, ist der Programmierer gezwungen, für jede noch so kleine Eingabeoperation einen ganzen Bildschirm voller Kommentartexte zu schreiben, der eine möglichst optimale Bedienerführung gewährleistet. In solchen Programmen kommen oft drei, vier oder noch mehr write oder writeln-Befehle auf jeden anderen. Bei der Austestung dieser Programme sind die Kompilierzeiten dann wieder so groß, daß der Turbo-Effekt nicht mehr zur Geltung kommen kann. Durch die vielen Ausgabebefehle wird der Quelltext so gewaltig, daß er in eine Vielzahl von Include-Files aufgegliedert werden muß. Das Editieren wird schon bei kleinsten Änderungen sehr mühsam.

Liegt ein solcher Fall vor, kann die Lösung nur darin liegen, die Bildschirm-Darstellungen für die Zeit der Programmerarbeitung völlig vom übrigen Programm zu trennen. Ist das Programm fertig, kann man sie dann wieder in den Bestand des Programmtextes aufnehmen. In diesem Abschnitt wird nun eine solche Vorgehensweise dargestellt. Besonders günstig ist die maschinelle Einarbeitung der Bildschirmdarstellungen in den Programmtext. Ein Nachteil ist der etwas langsamere Bildaufbau in der Testphase des Programms. Dieser Nachteil kann aber sicher toleriert werden.

Das Prinzip der Lösung besteht in der Auslagerung der Bildschirmdarstellungen in eine Diskettendatei. Jeder Bildschirm hat in dieser Datei eine Kennung, unter der er aufgefunden werden kann. Der Aufruf der Bildschirmdarstellungen erfolgt über eine Prozedur, der diese Kennung übergeben wird. In Bild 23 wird ein Auszug aus einer solchen Bildschirmdatei angegeben. Die Prozedur BildWrite (Bild 24), welche die Bildschirmdarstellungen aus der Datei liest und auf den Bildschirm bringt, realisiert folgende Einzelaktionen. Als erstes wird der Bildschirm gelöscht und die Bildschirmdatei auf ihren Anfang zurückgesetzt. Anschließend werden solange Zeilen aus dieser Datei gelesen, bis die zu suchende Kennung des zu schreibenden Bildes gefunden wird. Die folgenden Zeilen in der Datei enthalten nun das gesuchte Bildschirmaussehen. Diese Zeilen werden nun einzeln aus der Datei gelesen und auf den Bildschirm geschrieben. Als Endekennzeichen eines Bildes dient dabei die Kennung des nächsten Bildes oder das physische Dateiende. Wurde ein Bild gesucht, welches nicht in der Datei enthalten ist, erhält der Bediener eine entsprechende Mitteilung.

Durch die Hilfe dieser Prozedur genügt es jetzt, im Programm nur eine Anweisung zu schreiben, um dem Bildschirm das in der Datei gespeicherte Aussehen zu geben. Da diese Lösung nur dann angewendet wird, wenn in der Datei eine Vielzahl von Bildern gespeichert werden muß, ist die Datei natürlich auch sehr groß. Das heißt, die Suchzeiten für ein Bild sind teilweise recht lang. Der Vorteil liegt in der Tatsache, daß im Programm die Anweisung

**bildwrite('Plausibilität');**

genügt, um dem Bildschirm das gewünschte Aussehen zu geben. Weiterhin ist es nun auch möglich, ohne Umschreiben und Neu-Kompilieren des Programmes, die Textdarstellungen zu ändern. Es ist sogar denkbar, die kommentierenden Texte von den ersten Nutzern des Programms, die ja die Spezialisten auf ihrem Gebiet sind, ändern zu lassen, was mit jedem Textverarbeitungssystem möglich ist, da die Bilddatei ein reines Textfile ist. Sind die Tests des Programmes abgeschlossen, sollte diese Bildschirmdatei natürlich wieder in den Programmtext aufgenommen werden, um eine kompakte Einheit der Lösung zu erreichen. Dateien, an deren Namen sich später niemand mehr erinnert, werden leicht gelöscht, und unser Programm wird zum Invaliden. Dazu sind zwei Maßnahmen notwendig. Zum einen muß unsere Bilddatei in eine Pascal-Quelldatei umgewandelt werden, die in das Programm hineincompiliert werden kann. Zum zweiten muß die Prozedur bildwrite geändert werden. Diese mühselige Arbeit kann uns aber der Rechner abnehmen. Wir schreiben uns dazu ein Programm, das diesen Quelltext schreibt. Das Programm Transformation (Bild 25) liest unsere alte Bilddatei und erstellt uns sowohl die neue Prozedur bildwrite (Bild 26) als auch die Prozeduren, die später die einzelnen Bildschirmdarstellungen erstellen sollen (Bild 27). Die Namen der neuen Prozeduren werden in diesem Programm aus den alten Kennungen gebildet. Um Konflikte mit eventuell bereits bestehenden Bezeichnern zu vermeiden, wird den Prozedurnamen als erstes Zeichen noch ein **x** vorangestellt. Die Funktion des Programms Transformation ersieht man am einfachsten aus den gebildeten neuen Dateien.

Die Datei mit dem Namen BILDWRIT.INC enthält nach diesem Programm die neue Prozedur bildwrite (Bild 26). In der zweiten von unserem Programm erstellten Datei mit dem Namen NDEF000.INC sind nun alle die Prozeduren enthalten, die von der neuen Prozedur bildwrite aufgerufen werden (Bild 27). Die Tatsache, daß diese Prozeduren alle als Overlay-Prozeduren gekennzeichnet wurden, liegt an der Spezifik des hier genutzten Programmbeispiels.

```
In dieser Datei sind alle Bildschirm Masken des Programmes NDEF
enthalten.
Jede Maske beginnt mit ihrer Kennung :    >>>aaaaaaaa
Diese Kennung muss linksbuendig am Anfang jeden Bildes stehen!
>>>Hauptmenue

 ------------------------------------
!          H a u p t m e n u e       !      (= N D E F =)
!************************************!      ======
!BE-Typen     neu  definieren   !  1 !!-------------------------------
!             alte aendern      !  2 !!       S t e u e r u n g      !
!BE auf LP    neu  setzen       !  3 !!------------------------------!
!             verschieben       !  4 !!  Umfang der Plausibilitaets- T
                                          tests voreinstellen        Tr!
  ....
  ....
>>>Plausibilitaet
Folgende Stufen der Plausibilitaetstests sind moeglich:

Es erfolgt keinerlei Pruefung (schnellste Mode )     :       0

Pruefung bei Setzen ob BE-Referenzpunkt auf LP       :       1
   ...
   ...
>>>Freiflaechen
                             ---------------------------------------
                             !     Links/unten     !   Rechts/oben   !
Hier koennen Aussparungen    !---------------------!-----------------!
auf der Leiterplatte defi-   !____X____!____Y_____!____X____!____Y___!
  ....
```

◀ *Bild 23 Datei mit Bildschirmausschriften*

```
procedure BildWrite(suchtxt:string80);
  var zeile:string80;
  begin
    clrscr;                                  (* Bildschirm loeschen  *)
    reset(BildDatei);                        (* Bilddatei auf Anfang *)
    repeat
      readln(BildDatei.zeile)                (* Suchen der Kennung   *)
    until (zeile='>>>'+suchtext) or (eof(BildDatei);
    if not(eof(BildDatei)) then
      begin
        readln(BildDatei.zeile);
        repeat                               (* Beschreiben des Bild-  *)
          writeln(zeile);                    (* schirmes aus der Datei *)
          readln(BildDatei,zeile)
        until (copy(zeile,1,3)='>>>') or (eof(BildDatei))
      end
    else write('Nicht gefunden');
  end;
```

*Bild 24 Prozedur zum Schreiben der Bildschirmausschriften*

```
program Transformation;
const Hochkomma = #39 ;
var BildDatei,NeuInc,NeuBiW:text;
    zeile:string[80];
begin
  assign(BildDatei,'ndeftxt.txt');reset(BildDatei);
  assign(NeuInc,'ndef000.inc');rewrite(NeuInc);
  assign(NeuBiW,'b:'ersat.inc');rewrite(NeuBiW);
  writeln(NeuBiW,'procedure bildwrite(suchtxt:string80);     (* Neuer     *)
  writeln(NeuBiW,'  begin');                                 (* Programm  *)
  writeln(NeuBiW,'    clrscr;');                             (* Kopf      *)
  repeat
    readln(BildDatei,zeile)        (* Suchen nach erstem Bildschirm-Bild *)
  until copy(zeile,1,3)='>>';
  repeat
    zeile:=copy(zeile,4,length(zeile)-3);
    writeln(NeuBiW,
          if suchtxt='+Hochkomma+zeile+Hochkomma+' then x'+zeile+';');
    writeln(NeuInc,'overlay procedure x'+zeile+';');
    writeln(NeuInc,' begin');
    writeln('procedure x'+zeile);   (* Meldung ueber Programmlauf *)
    readln(BildDatei,zeile);
    repeat
      writeln(NeuInc,'  writeln('+Hochkomma+zeile+Hochkomma+');');
      readln(BildDatei,zeile)
    until (copy(zeile,1,3)='>>') or eof(BildDatei);
    writeln(NeuInc,'end;')
  until eof(BildDatei);
  writeln(NeuBiW,'  end;
  close(BildDatei);
  close(NeuInc);
  close(NeuBiW);
end.
```

Bild 25 *Quelltext-schreibendes Programm*

```
overlay procedure xHauptmenue;
 begin
  writeln('..................................');
  writeln('|            H a u p t m e n u e      |   +- N D E F -+');
  writeln('|^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^|   ---------      ');
  writeln('|BE-Typen     neu  definieren  | | | |   S t a r t i n g ');
    ...
   ....
 end;
overlay procedure xPlausibilitaet;
 begin
  writeln('');
  writeln('Folgende Arten der Plausibilitaetstests sind moeglich:');
  writeln('^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^');
   ....
   ...
 end;
 ....
 ....
```

Bild 27 *Automatisch erstellter Quelltext*

```
procedure bildwrite(suchtxt:zeilentyp);
 begin
   clrscr;
    if suchtxt='Hauptmenue' then xHauptmenue;
    if suchtxt='Plausibilitaet' then xPlausibilitaet;
    if suchtxt='BETypendefinieren' then xBETypendefinieren;
       ....
    if suchtxt='Freiflaechen' then xFreiflaechen;
       ....
    if suchtxt='Anzeigen' then xAnzeigen;
 end;
```

Bild 26 *Neue Prozedur BildWrite*

```
procedure List_Hardcopy;
  var Bild : array[1..24,1..80] of char absolute $F800;
      i,j  : byte;
  begin
    for i:=1 to 24 do
      begin
        for j:=1 to 80 do write(lst,Bild[i,j]);
        writeln(lst);
      end;
  end;
```

Bild 28 *Die einfache Hardcopy-Prozedur*

Hat der Programmierer das Programm *Transformation* auf seine Textdatei mit den Bildschirmdarstellungen angewendet, sind nur noch wenige Schritte notwendig. Als erstes tauscht er die alte Prozedur BildWrite gegen die neue aus, was mit dem Editor-Kommando ^KR (Block-Lesen) kein Problem ist. Dann fügt er im Programm die Compiler-Anweisung (* **$I ndef000.inc** *) ein. Nach der nun notwendigen Neuübersetzung des Programmes ist alles erledigt.

Als letztes sollte noch gesagt werden, daß dieses Programm nur *ein* Beispiel ist. In vielen Situationen ist es möglich, stupide Programmierarbeit von kleinen, selbstgeschriebenen Hilfsprogrammen erledigen zu lassen. Wem dieser Gedanke vertraut geworden ist, dem ergeben sich in der täglichen Arbeit oft solche Möglichkeiten.

### Daten – und Dateiarbeit

In diesem Kapitel soll es sowohl um die Bearbeitung von Daten aus Turbo-Pascal-fremden Diskettenfiles als auch um die Verwaltung von Daten in listenartigen Strukturen im Hauptspeicher gehen. Neben diesen größeren Problemen soll hier aber auch Platz für einige kleinere sein. So wird eine Möglichkeit geboten, das Verzeichnis einer Diskette selektiv zu lesen.

### Hardcopy

Hat ein Rechner bereits eine Hardcopy-Funktion, so wird diese oft und gern genutzt. Der Griff zur Taste PrnScr (PrintScreen) wird zu einem wertvollen Hilfsmittel in der Arbeit. Vom Bildschirm schnell und problemlos ein interessantes Aussehen auf dem Papier festhalten zu können, effektiviert die Arbeit an einem Rechner doch sehr. Hat man sich an einem Rechner höherer Leistungsklasse an dieses Hilfsmittel gewöhnt, vermißt man es auf allen anderen. Hier möchte ich nun zwei Prozeduren anbieten, die diesen Service auch auf kleineren Rechnern ermöglichen und speziell für den PC 1715 entwickelt wurden. Für ihre Lauffähigkeit sind ähnliche Einschränkungen wie für die Fensterhilfen gültig. Der Bildwiederholspeicher muß linear von der Hauptspeicheradresse F800H an aufgebaut sein, und jede Position auf dem Bildschirm muß genau einer Speicherzelle entsprechen. Dies ist zum Beispiel nicht mehr der Fall, wenn Bildschirmsteuerzeichen wie im Kapitel Auswahlmenü Speicherplatz beanspruchen. Eine Sicherung von Bildschirmen mit intensiven oder inversen Bereichen ist also nicht möglich.

Als erstes wird die Prozedur List_Hardcopy vorgestellt, die das aktuelle Aussehen des Bildschirmes ausdruckt (Bild 28).

Die Wirkungsweise dieser Prozedur beruht in der Überlagerung des Arrays *Bild* mit dem Bildwiederholspeicher des Rechners. Die Ausgabe des Bildschirmes auf den Drucker erfolgt zeilenweise, Zeichen für Zeichen. Diese Prozedur beansprucht in unseren Programmen kaum Platz. Konkret sind es 10 Byte Daten und 125 Byte Code.

Wir wollen aber noch eine zweite, komfortablere Hardcopy-Prozedure vorstellen, mit der es auch möglich ist, den Bildschirminhalt in eine Diskettendatei zu schreiben, wodurch er gesichert wird und dann jederzeit wieder zu Papier gemacht werden kann. Diese Programmlösung wird auch deshalb angeführt, da in ihr gut zu erkennen ist, wie man das Standardtextfile LST als File-Variable behandeln kann. Dieses Prinzip ist auch in vielen anderen Situationen interessant. Selbstverständlich benötigt diese Variante mehr Platz. Sie belegt genau 283 Byte Daten und 422 Byte Code. (Bild 29)

Die prinzipielle Wirkungsweise der Prozedur List_Hardcopy finden wir hier in der lokalen Prozedur AusgabeAuf wieder. Worauf diese Ausgabe erfolgen soll, wird der Prozedur als Parameter übergeben. Dieser Parameter spezifiziert ein beliebiges Text-File. Auch der Drucker, also das Standard-File LST, ist ein Text-File.

Die Prozedur hardcopy sichert zuerst das Aussehen der ersten Bildschirmzeile in einem temporären Feld. Dann erfolgt auf dieser ersten Bildschirmzeile die Frage nach dem Namen des Diskettenfiles, unter dem die Hardcopy abgelegt werden soll. Anschließend wird das Aussehen der ersten Zeile wieder hergestellt und der Kursor ausgeschaltet. Ist der eingegebene Name leer, daß heißt, die Frage wurde nur mit der Entertaste beantwortet, so erfolgt die Ausgabe wie gewohnt über den Drucker. Ansonsten wird die angewiesene Datei zum Schreiben vorbereitet und gefüllt. Zum Abschluß muß natürlich der Kursor wieder eingeschalten werden.

```
procedure hardcopy;
  const Kursein = #$82;
        Kursaus = #$83;
  var Bild:array[1..24,1..80] of char absolute $F800;
      i,j :byte;
      Temp:array[1..80] of char;
      Dateiname:string[14];
      Diskfile :text;
  procedure AusgabeAuf(var Geraet:text);
    begin
      for i:=1 to 24 do
        begin
          for j:=1 to 80 do write(Geraet,Bild[i,j]);
          writeln(Geraet);
        end;
      close(Geraet)        (* close auf lst bleibt ohne Wirkung *)
    end;
  begin
    for i:=1 to 80 do          (* Sichern der ersten Zeile des BS  *)
      Temp[i]:=Bild[1,i];      (* in ein temporaeres Feld          *)
    gotoxy(1,1);
    write('Auf Datei (RET fuer Drucker)');
    read(Dateiname);           (* Lesen Filenamens unter dem die  *)
    write(kursaus);            (* Hardcopy gespeichert werden soll *)
    for i:=1 to 80 do
      Bild[1,i]:=Temp[i];      (* Wiederherstellen erste BS Zeile  *)
    if dateiname<>'' then      (* Test des Dateinamens *)
      begin
        assign(Diskfile,dateiname);
        rewrite(Diskfile);
        AusgabeAuf(Diskfile)   (* Ausgabe auf Diskfile  *)
      end
    else AusgabeAuf(lst);      (* Ausgabe auf Drucker   *)
    write(kursein);
  end;
```

```
type string11=string[11];
procedure MiniDir(LW:Char;Maske:string11);
  var fcb  : array[0..35] of char absolute $5C;
      dma  : array[0..127] of byte;
      a,i  : byte;
  procedure Eine_Eintragung(DMA_Position:byte);
    var Name:string[12];
    begin
      DMA_Posi:=DMA_Pos*32+1; Name:='';
      for i:=0 to 11 do
        Name:=Name+chr(dma[DMA_Pos+i] and $7F);
      insert('.',Name,9);
      write(Name,' ! ')
    end;
  begin
    LW:=upcase(LW);
    bdos(#1A,addr(dma)) ;                      (* Einstellen DMA Adresse   *)
    fcb[0]:=chr(integer(LW)-integer('A')+1);   (* Einstellen des Laufwerks *)
    [illegible]
    a:= bdos(#11,addr(fcb)) ;                  (* Suchen nach erster Eintragung *)
    while a < 255 do
      begin
        Eine_Eintragung(a) ;
        a:= bdos(#12,Addr(fcb)) ;              (* Suchen weiterer Eintragungen *)
      end;
    writeln;
  end;
```

**Bild 31 Kleines Disketten-DIRectory MiniDir**

**◂ Bild 29 Hardcopy wahlweise auf Drucker oder Datei**

Existiert bereits eine Disketten-Datei unter dem angegebenen Namen, so ist deren ursprünglicher Inhalt gelöscht. Da dieses Problem keinesfalls spezifisch für diese Prozedur ist, wollen wir eine kleine Funktion angeben, die abprüft, ob eine Datei bereits existiert. Jeder Programmierer kann sie nutzen, wenn Dateinamen überprüft werden müssen (Bild 30).

Wollen wir die Dienste der Funktion Exist in Anspruch nehmen, sieht der Test des Dateinamens in obiger Prozedur folgendermaßen aus:

```
...
if (dateiname<>'')and
   (not(exist(dateiname)) then
...
```

### Diskettenverzeichnisse

In Fortführung des letzen Abschnittes wollen wir noch ein wenig bei den Namen von Dateien bleiben. Wird der Nutzer eines Programmes aufgefordert, einen Dateinamen anzugeben, ist dieser meistens etwas allein gelassen. Kann er sich noch an den Namen *seiner* Datei erinnern, so nicht mehr an die Extension. Weiß er diese noch, fehlt ihm der Name. Oder aber er weiß im Augenblick keines von beiden. Der Programmierer sollte an diesen Programmstellen Unterstützung vorsehen. Bei Compilern, die den Systembefehl DIR unterstützen, ist dies leicht möglich. Hier nun sollen zwei Prozeduren auch für die kleineren Compiler vorgestellt werden. Sie sind auf allen CP/M-kompatiblen Betriebssystemen lauffähig, da sie die Diskettenzugriffe über BDOS-Funktionen realisieren. Als erstes stellen wir die Prozedur MiniDir vor. Eine genaue Erläuterung der Wirkungsweise der verwendeten BDOS-Rufe muß hier entfallen, da dies den Rahmen des Kurses sprengen würde. Es sei nur erwähnt, daß das Array **fcb** einen File-Control-Block darstellt. Dieses Array wird vor allem dazu genutzt, dem Betriebssystem Informationen zu übergeben. Im Array **dma** finden wir nach den BDOS-Rufen die angeforderten File-Namen. Die BDOS-Funktionen selbst übergeben lediglich eine Information, wo im Array dma der neue File-Name zu finden ist, da im dma-Block bis zu vier Namen gleichzeitig geführt werden können. Haben diese Funktionen keine weitere Verzeichniseintragung gefunden, übergeben sie den Wert 255 an das Programm. Der Anweisungsteil der Prozedur MiniDir stellt die Vorgabewerte des Arrays fcb ein und sucht nach den Verzeichniseintragungen. Die lokale Prozedur Eine_Eintragung, der die Position des neuen File-Namens übergeben wird, schreibt diesen Namen auf den Bildschirm (Bild 31).

Der Aufruf der Prozedur MiniDir erfordert eine gewisse Sorgfalt. Es handelt sich wirklich nur um ein *Mini*-Directory. Werden nicht vorhandene Laufwerke im Prozedure-Kopf angegeben, kommt es zu Systemabstürzen. Auch die Maske muß mit genau 11 Zeichen aufgefüllt werden. Sie stellt eine Gültigkeitsmaske für das Suchen nach Verzeichniseintragungen dar. Die ersten 8 Zeichen selektieren die Dateinamen, das 9., 10. und 11. Zeichen selektieren die Extension. Ein Punkt darf in der Maske nicht vorkommen. Für beliebige Zeichen ist je ein "?" einzutragen. Soll die Länge der Namen begrenzt werden, so sind die restlichen Zeichen mit Leerzeichen aufzufüllen. In Bild 32 sind einige Beispiele für den Aufruf der Prozedur MiniDir angegeben.

Die Prozedur MiniDir listet die gefundenen Verzeichniseintragungen von der aktuellen Kursorposition aus auf. Es werden alle in die Maske passenden Dateien aufgelistet, also auch Dateien mit den Attributen SYS und R/O. Stand der Kursor an einem Zeilenanfang, stehen immer 5 Verzeichniseintragungen nebeneinander.

Als nächstes wird eine Lösung vorgestellt, die viele der aufgeführten Unzulänglichkeiten nicht mehr aufweist. Die Angabe des Parameters VerzMaske der Prozedur DIR kann nun dem Nutzer überlassen werden (Bild 33).

Die Prozedur DIR selbst bearbeitet und interpretiert die angegebene Verzeichnismaske und selektiert, falls angegeben, die Laufwerksangabe. Fehlt diese, wird das aktuelle Laufwerk ermittelt. Die Wirkungsweise der Funktion HoleDirectory entspricht der der Prozedur MiniDir. Sie übergibt die Anzahl der aufgefundenen Verzeichniseintragungen. Die Angabe der Verzeichnismaske kann nun recht frei erfolgen. Zwischen kompletten Dateinamen mit Laufwerksangabe und einer völlig leeren Zeichenkette ist alles erlaubt. Die Interpretation der Verzeichnismaske in dieser Prozedur weicht gewollt von der Interpretation der Maske beim Systembefehl DIR ab. So liefert zum Beispiel der Aufruf:

**DIR(a:TP.A);**

alle die Dateien, deren Name mit TP. und deren Extension mit A beginnt. Dies war in der konkreten Anwendung, der diese Lösung entnommen wurde, gefordert. Ist in einer anderen Anwendung eine andere Interpretation gewünscht, kann dies leicht geändert werden. Im Gegensatz zum MiniDir werden die Verzeichniseintragungen in dieser Lösung nicht nur auf den Bildschirm übertragen, son-

```
function exist (filename:string[14]):boolean;
  var tfile:file;
  begin
    assign(tfile,filename);
    {$I-};
    reset(tfile);
    {$I+};
    if ioresult <> 0 then
      exist:=false
    else exist:=true;
    close(tfile);
  end;
```

**Bild 30 Test auf Existenz einer Datei**

```
MiniDir('A','????????ASC');   (* alle Dateien mit Ext. ASC auf LW A         *)
MiniDir('B','TURBO??????');   (* Dateien die mit TURBO beginnen Ext. bel.*)
MiniDir('A','TURBO   ???');   (* Dateien Namens TURBO mit bel. Extension *)
MiniDir('A','???     ASC');   (* Dateien mit hoechstens dreistelligem      *)
                              (* Namen und Extension ASC auf LW A          *)
MiniDir('B','???????????');   (* alle Dateien auf LW B                     *)
```

*Bild 32 Aufruf-Beispiele für MiniDir*

```
type string11=string[11];string14=string[14];
var DateiNamen : array[1..128] of string[12];
    Datei_Anzahl: byte;
function HoleDirectory(LW:Char;Maske:string11):byte;
  var fcb  : array[0..35] of char absolute Maske;
      dma  : array[0..127] of byte;
      a,i,zaehler  : byte;
  procedure Eine_Eintragung(DMA_Posi:integer);
    begin
      DMA_Posi:=DMA_Posi*32+1 ;
      if (dma[DMA_Posi+10] div 128 =0)              (* Test auf gesetzte    *)
         and(dma[DMA_Posi+11] div 128 =0)  then (* Atribute Sys und R/O *)
        begin
          DateiNamen[zaehler]:='';
          for i:=0 to 11 do
            DateiNamen[Zaehler]:=
            DateiNamen[Zaehler]+chr(dma[DMA_Posi+i] mod 128);
          insert('.',DateiNamen[Zaehler],9);
          zaehler:=succ(zaehler)
    end end;
  begin
    zaehler:=1 ;LW:=upcase(LW);
    bdos($1A,addr(dma)) ;                (* Einstellen DMA-Adresse     *)
    fcb[0]:=chr(ord(LW)-ord('A')+1)   (* Einstellen des Laufwerkes *)
    fcb[12]:= #0;
    a:= bdos($11,addr(fcb)) ;     (* Suchen nach erster Eintragung *)
    while a<255 do
      begin
        Eine_Eintragung(a);
        a:= bdos($12,addr(fcb)) ; (* Suchen weiterer Eintragungen *)
      end;
    HoleDirectory:=zaehler-1 ;
  end;
procedure DirWrite(Anzahl:byte);
  var n: byte;
  begin
    for n:=1 to Anzahl do  write(n:3,': ',DateiNamen[n],' ! ');
  end;
procedure Dir(VerzMaske:string14);
  var i:byte;LW:char;mask:string11;
  begin
    while pos('*',VerzMaske)<>0 do delete(VerzMaske,pos('*',VerzMaske),1);
    if (VerzMaske[2]=':')and(length(VerzMaske)>1) then
      LW:=upcase(VerzMaske[1])                          (* Laufwerk angegeben  *)
    else
      begin
        LW:=char(bdos($19)+ord('A'));                 (*  Standart-Laufwerk  *)
        VerzMaske:=lw+':'+VerzMaske
      end;
    if length(VerzMaske)=2 then mask:='???????????'   (* Keine Maske  *)
    else
      begin
        while (pos('.',VerzMaske)<11)and(pos('.',VerzMaske)<>0) do
          insert('?',VerzMaske,pos('.',VerzMaske));(* Auffuellen Dateinamen *)
        delete(VerzMaske,11,1);                        (* Loeschen des Punktes  *)
        while length(VerzMaske)<14 do
          VerzMaske:=VerzMaske+'?';                    (* Auffuellen Extension  *)
        for i:=1 to 11 do mask:=mask+upcase(VerzMaske[i+2]);
      end;
    Datei_Anzahl:=HoleDirectory(LW,mask);
    DirWrite(Datei_Anzahl)
  end;
```

*Bild 33 Eine große DIRectory-Variante*

```
procedure ClearDirWrite4(Anzahl:byte);
  var x,y,n: byte;
  begin
    clrscr;                                            (* Loeschen Bildschirm  *)
    for n:=1 to Anzahl do                  (* 4 Eintragungen auf einer Zeile *)
      begin
        x:=(n-1) mod 4*19+1;                        (* 4 x 19 Zeichen je Zeile *)
        y:=(n-1) div 4+4;                                (* erste Zeile  = 4 *)
        gotoxy(x,y);
        write(n:3,':',DateiNamen[n],' !');             (* Mit Numerierung *)
      end;
    writeln
  end;
procedure DirWrite5(Anzahl:byte);
  var n: byte;
  begin
    for n:=1 to Anzahl do
      write(DateiNamen[n],' !   ');   (* 5 Eintragungen auf einer Zeile *)
    writeln
  end;
procedure DirWrite6(Anzahl:byte);
  var n: byte;
  begin
    for n:=1 to Anzahl do
      begin
        write(DateiNamen[n],'   ');     (* 6 Eintragungen auf einer Zeile *)
        if n mod 6 = 0 then writeln
      end;
    writeln
  end;
```

*Bild 34 Weitere Druckvarianten eines Verzeichnisses*

```
type PathName = string[80];
     DirName  = string[12];
var  FCB  : Array[-7..35] of char;
     Regs : record case boolean of
              true :(ax,bx,cx,dx,bp,si,di,ds,es,Flags :integer);
              false:(al,ah,bl,bh,cl,ch,dl,dh              :byte   )
            end;
     DMA  : record
              FCB       :array [0..$14] of char: (* nur intern *)
              Attribut :byte;
              Zeit,Datum,SizeLo,SizeHi:integer;
              Filename :array[1..13] of char
            end;

function ErsterEintrag(Name:Pathname;Attr:byte):Dirname;
  begin
    with Regs do begin
      Name:=Name+#0 ;        ah:=$1A;
      ds  :=Seg(DMA);        dx:=Ofs(DMA);
      MsDOS(Regs);
      ah  :=$4E;             cx:=Attr;
      ds  :=Seg(Name[1]);  dx:=Ofs(Name[1]);
      MsDOS(Regs);
      if odd(Flags) then ErsterEintrag:=''
        else with DMA do
          ErsterEintrag:=copy(Filename,1,pred(pos(#0,Filename)))
  end end;
function NaechsterEintrag:Dirname;
  begin
    with Regs do begin
      ah:=$4F;
      MsDOS(Regs);
      if odd(Flags) then ErsterEintrag:=''
        else with DMA do
          ErsterEintrag:=copy(Filename,1,pred(pos(#0,Filename)))
  end end;
Procedure dir(name:Pathname);
  var n:Pathname;
  begin
    n:=erstereintrag(name,0);
    while n<>'' do begin
      write(n,' ':16-length(n));
      n:=NaechsterEintrag
  end end;
```

*Bild 35 DIRectory unter MS-DOS*

dern auch im Array DateiNamen gesammelt. Somit stehen sie einer weiteren Auswertung zur Verfügung. Dateien, die die Attribute R/O und SYS besitzen, werden diesmal überlesen. Weiter wurde die Ausgabe auf den Bildschirm vom Lesen des Verzeichnisses selbst getrennt. Dadurch begünstigt ist es nun leicht möglich, das Ausgabeformat des Verzeichnisses zu ändern. Die Prozedur DirWrite, mit der im obigen Beispiel die Ausgabe auf dem Bildschirm vorgenommen wird, gestattet auch eine Durchnumerierung der aufgelisteten Dateien und zeigt jeweils vier Dateien in einer Zeile an. In Bild 34 werden noch drei weitere Verzeichnis-Ausgabe-Prozeduren angeboten, die jeweils ein anderes Format auf dem Bildschirm erzeugen.

Weder die Prozedur MiniDir, noch die Prozedur DIR, werden sofort und ohne Änderungen Aufnahme in Programme finden, da die Ansprüche an die speziell zu erbringende Leistung doch sehr unterschiedlich sein können; mit kleinen Änderungen werden sie aber sicher vielen Anforderungen gerecht.

Da die Auswertung eines Diskettenverzeichnisses natürlich nicht nur unter CP/M-kompatiblen Betriebssystemen interessant ist, wird in Bild 35 noch eine entsprechende Lösung für die Arbeit unter MS-DOS gezeigt. Da die prinzipielle Wirkungsweise den Lösungen unter CP/M entspricht, erübrigt sich eine nähere Beschreibung. Zu erwähnen ist lediglich, daß der Name, der der Prozedur dir übergeben wird, neben der Dateimaske sowohl Laufwerks- als auch Pfadangaben enthalten darf.

*wird fortgesetzt*

# Turbo-Pascal-Praxis

**Teil 4**

**Manfred Zander, Dresden**

*Nachdem wir uns im Teil 3 unseres Kurses mit Bildschirmdarstellungen, Hardcopyfunktionen und Disketteninhaltsverzeichnissen beschäftigt haben, wenden wir uns in dieser Folge der Bearbeitung von Dateien anderer Programme zu und werden die Verwaltung von Daten als Listen oder als Bäume beginnen.*

**Dateien anderer Programme**

Bei der Programmierung ist es oft notwendig, eine Datenschnittstelle zu anderen Programmsystemen zu gewährleisten. Der häufigere Fall ist dabei die Übernahme von Ergebnissen aus anderen Programmen. Sind diese Programme nicht in Turbo-Pascal geschrieben, kann dies auf Grund der verschiedenen Dateistrukturen ein Problem werden. Ein oft genutzter Weg ist dann die Übertragung der Daten mit Hilfe von Textdateien. Das bedingt aber immer einen erhöhten Aufwand an Datenkonvertierungen. Entweder müssen im erstellenden Programm diese Text-Dateien für das lesende Programm extra aufgebaut werden, oder aber man muß spezielle Konvertierungsprogramme entwikkeln, um aus binären Dateien nur für die Übertragung Textdateien erstellen zu können. Dabei ist im lesenden Programm dann aber die Rücktransformation in bearbeitbare Daten natürlich auch wieder notwendig.

Typische Fälle sind die Übertragung von dBase II-Datenbankdateien in ein Turbo-Pascal-Programm sowie die Bearbeitung von Bilddateien, die von CAD-Arbeitsplätzen erstellt wurden. In beiden Fällen besitzen die Dateien, die gelesen werden sollen, eine Dateistruktur, die nicht sofort lesbar ist. Da auch bei der File-Arbeit die strenge Typüberwachung im Compiler wirksam ist, gelingt es in den seltensten Anwendungen, diese Klippe durch eine ausgeklügelte Datei-Typisierung zu umschiffen. Es gibt nur zwei Dateitypen, die auf beliebige Dateien angewandt werden können. Dies sind der vordeklarierte Typ **file**, mit dem nur blockweise gearbeitet werden kann und unter einigen Betriebssystemen der Typ **file of char** bzw. **file of byte**.

Prinzipiell ist noch zu sagen, daß vor jeder Erarbeitung einer solchen Lösung eine intensive Analyse der zu lesenden Datendateien notwendig ist. Erst die so gewonnenen Informationen bieten dann die Möglichkeiten zur Lösung der Aufgaben.

*dBase II-Dateien*

Die Struktur der Datenbankdateien von dBase II ist nicht vollständig durch eine Typvereinigung unter Turbo-Pascal beschreibbar. Diese Tatsache resultiert vor allem daraus, daß die Dateien aus einem Dateikopf bestehen, der völlig anders strukturiert ist als die folgenden Teile der Datei, in der die Daten gespeichert sind. Die Struktur des Dateikopfes ist bei allen Datenbankdateien dieses Programmsystems gleich und kann unter Turbo-Pascal durch einen kombinierten Record- und Array-Typ beschrieben werden. In diesem Kopf einer konkreten Datenbankdatei stehen dann die Informationen darüber, aus wievielen und welchen Feldern die folgenden Datensätze aufgebaut sind. Auf die Angabe einer konkreten Realisierung zum Lesen und Schreiben von DBF-Dateien kann hier verzichtet werden, da es bereits eine ausreichende Anzahl von Veröffentlichungen zu diesem Thema gibt.

*CAD-Bilddateien*

In den letzten Jahren hat die Verbreitung von CAD-Grafikarbeitsplätzen rapide zugenommen. Diese neuen Arbeitsplätze erlauben es, Konstruktionsarbeiten am Grafikbildschirm mit bedeutend verringertem Zeitaufwand qualitativ hochwertig zu erledigen. Das primäre Ergebnis einer so durchgeführten Konstruktion ist aber immer eine Datei, in der die erarbeitete Grafik gespeichert ist. Was nun fehlt, ist ein Bindeglied zwischen dieser Datei und der jeweils vorhandenen Technik und Technologie. Ein sofortiger Austausch dieser Technik ist in den seltensten Fällen ökonomisch sinnvoll oder möglich. Es besteht also das Bedürfnis, die auf den Grafikarbeitsplätzen entstehenden Bilddateien maschinell lesen zu können, um eine Umwandlung der enthaltenen Informationen in die vorgeschriebenen Datenformate erreichen zu können. Man kann zwar die Konstruktionsergebnisse der neuen Systeme plotten und diese geplotteten Bilder dann wieder digitalisieren, aber zum Standard sollte dieses Vorgehen nicht erhoben werden. Es muß eine Möglichkeit der maschinellen Datenübertragung zwischen den verschiedenen Systemen gefunden werden. Diese Übertragung sollte in beide Richtungen vorgesehen werden. Daher ergibt sich also auch die Aufgabe, die Bilddateien der neuen Systeme schreiben zu können. Solche Lösungen können selbstverständlich auch dazu genutzt werden, Funktionen zu entwickeln, die das CAD-System selbst nicht bietet. Dabei wird dann eine Bilddatei des Systems gelesen und der bearbeitete Stand dieser Datei im gleichen Format wieder auf Diskette oder Festplatte abgelegt.

Als Beispiel wird im folgenden der binäre Datenaufbau des sehr weit verbreiteten CADdy 3.04 erläutert und eine Lösung für das Lesen und Schreiben der Bilddateien, dieses auf 16-Bit-Technik laufenden CAD-Systems geboten. Wie das genannte CAD-System selbst, erfordert auch die vorgestellte Lösung einen 16-Bit-Rechner und MS-DOS. Weiterhin muß diese Lösung mit dem speziellen Compiler Turbo-87 übersetzt werden. Dieser Compiler, der den Koprozessor 8087 unterstützt, gestattet die Berechnung der hochgenauen Realzahlen, die in der Lösung benötigt werden.

Die Struktur der Bilddateien des genannten Systems kann auf den ersten Blick betrachtet mit einer variablen Typvereinbarung (einer Datensatz-Variante) beschrieben werden. Jeder einzelne Grafiksatz, der beispielsweise eine Linie oder einen Punkt beschreibt, hat an seinem Anfang eine konstante Struktur. In dieser Kopfstruktur existiert jeweils ein Zeichen zur Kennzeichnung des folgenden Inhalts. In Abhängigkeit von diesem Zeichen enthält der Rest des Datensatzes alle notwendigen Informationen, um einen Punkt oder eine Linie zu beschreiben. Natürlich sind diese Informationen für verschiedene grafische Elemente auch verschieden aufgebaut und belegen jeweils eine unterschiedliche Anzahl Bytes. Leider ist es aber trotzdem nicht möglich, einen solchen kompakten Satztyp zu erstellen und die Bilddateien mit ihm zu typisieren. Dies hat vor allem zwei Gründe. Als erstes ist das Entscheidungszeichen im konstanten Satzkopf nicht die letzte Angabe im Kopf. Diese Klippe kann zwar umschifft werden, ein zweiter Grund macht dieses einfache Vorgehen aber unmöglich: Die einzelnen Datensätze sind in der Datei mit ihren verschiedenen Größen nahtlos aneinandergefügt. Damit ist eine Grundeigenschaft typisierter Files, die konstante Satzlänge, nicht mehr gegeben. Es bleibt also noch der Weg, die Bilddateien als *file of byte* zu behandeln.

Das Prinzip der im Bild 36 vorgestellten Lösung ist im Verhältnis zu ihrem Schreibaufwand einfach. Die Prozedur *Satzlesen* gestattet das Einlesen eines kompletten Datensatzes. Dazu liest sie zuerst byteweise den konstant typisierten Kopf eines Satzes in das Array *Kopfbuffer*. Dieses Feld ist der Variablen *Kopf*, die die wahre Struktur dieses Satzkopfes enthält, überlagert. Hier sind konstant ein Folgenkennzeichen des Grafikelementes, dessen Farbe und dessen Zeichenebene beschrieben. Für die weitere Organisation sind die folgenden Variablen *Was* und *Zahl* im Kopf von Bedeutung. *Was* gibt an, welche Art von Bildelement folgt, und in *Zahl* ist die Länge des restlichen Datensatzes gespeichert. Wenn in *Was* ein **V** steht, das heißt das Versionskennzeichen folgt, wird dieses abgetestet. Handelt es sich in *Was* um eine andere Kennung, wird der Rest des Satzes lediglich auf das Array *Buffer* aufgelesen. Um die Anzahl der hierbei zu lesenden Bytes zu ermitteln, sind von der Variablen *Kopf.zahl* lediglich 16 Byte zu substrahieren.

Dem Array *Buffer* sind alle möglichen Reststrukturen eines Grafik-Satzes überlagert. Das lesende Programm kann sich nun anhand der Variablen *Kopf.Was* informieren, welche Art von Bildelement eingelesen wurde und daraufhin die entsprechenden Informationen aus dem zugehörigen überlagerten Feld entnehmen. Beim nächsten Aufruf der Prozedur Satzlesen werden diese Daten durch den neuen Satz überschrieben. Die Tafel 1 gibt an, welche Kennungen für welche grafischen Elemente im Feld *Kopf.Was* genutzt werden. Die Prozedur Satzschreiben realisiert die Umkehrung des beschriebenen Vorganges. Sie speichert den grafischen Sachverhalt, der zuvor in den entsprechenden Feldern abgelegt sein muß, in eine Bilddatei. Dabei orientiert sie sich wieder an der

```
type Picfiletype = file of char;
var Kopf : record                              (* Feste Kopf-Struktur  *)
             folge,farbe,ebene:integer;        (* aller Graphik-Saetze *)
             Was:char; zahl :byte
           end;
    Kopfbuffer : array[1..8] of char absolute Kopf;
    Buffer:array[1..128] of char;
    Apuf : record
             y,x,w,zy,zx:real;name:string[8]
           end absolute Buffer;
    aapuf: record
             y,x:integer;Name:string[7]
           end absolute Buffer;
    Cpuf : record
             y,x,th:real; Ctb,Cta,Ctw:integer;
             ttext:string[15]; INT1,int2:real
           end absolute Buffer;
    ddpuf: record
             y,x,th:real;tb,ta,tw:integer;
             ttext:string[15]
           end absolute Buffer;
    Dpuf : record
             y,x,th:real;tb,ta,tw:integer;
             ttext:string[79]
           end absolute Buffer;
    Epuff :char absolute Buffer;
    Kpuf : record
             y,x,w2,w1:real;ls,lTyp:integer;r:real;
             lb,la:integer
           end absolute Buffer;
    Lpuf : record
             y1,x1,y2,x2:real; lTyp,ls:integer
           end absolute Buffer;
    mmpuf: record
             y1,x1,y2,x2,y3,x3:real
           end absolute Buffer;
    Mpuf : record
             y1,x1,y2,x2,y3,x3:real
           end absolute Buffer;
    Npuf : record
             y1,x1,y2,x2:real
           end absolute Buffer;
    Opuf : record
             y,x:real
           end absolute Buffer;
    Qpuf : record
             bgy,bgx:real
           end absolute Buffer;
    Rpuf : record
             zoomY,zoomX:real
           end absolute Buffer;
    Vpuf : record
             c1,c2,c3,c4:char; i1,i2,i3:integer
           end absolute Buffer;
    Xpuf : record
             Refy,Refx:real
           end absolute Buffer;

  procedure Satzlesen(var datei:Picfiletype);
    var i:integer;
    begin
     for i:=1 to 8 do read(datei,Kopfbuffer[i]);
     if Kopf.Was = 'V'
       then begin for i:=1 to 10 do read(datei,Buffer[i]);
                if (Vpuf.c1='3')and(Vpuf.c2='.') and
                   (Vpuf.c3='0')and(Vpuf.c4='4') then write(' 3.04 ')
                  else writeln('Datei hat nicht das Format von CADdy 3.04 ');
            end
       else for i:=1 to Kopf.zahl-16 do read(datei,Buffer[i]);
    end;

  procedure Satzschreiben(var datei:Picfiletype);
    var i:integer;
    begin
    with Kopf do
     case Was of
     'a':     zahl:=$1C;       'A':     zahl:=$41;
     'E':     zahl:=$10;       'K':     zahl:=$3C;
     'L':     zahl:=$34;       'M','m': zahl:=$40;
     'N':     zahl:=$30;       'V':     zahl:=$4;
     'd','D': zahl:=$1F+$10+length(ddpuf.ttext);
     'C':     zahl:=$2F+$10+length(Cpuf.ttext);
     'X','R','O','Q':zahl:=$20;
     end;
    for i:=1 to 8 do write(datei,Kopfbuffer[i]);
    if Kopf.Was = 'V'
       then begin
              Vpuf.c1:='3';Vpuf.c2:='.';Vpuf.c3:='0';Vpuf.c4:='4';
              Vpuf.i1:=0;Vpuf.i2:=0;Vpuf.i3:=0;
              for i:=1 to 10 do write(datei,Buffer[i]);
            end
       else for i:=1 to Kopf.zahl-16 do write(datei,Buffer[i])
    end;

Procedure write_L(var Datei:PicFileType;x,y,xx,yy:real;e,col:integer);
  begin with Lpuf do begin
    x1:=x;x2:=xx;y1:=y;y2:=yy;lTyp:=1;ls:=0 end;
    with kopf do begin ebene:=e;farbe:=col;folge:=0;Was:='L' end;
    satzschreiben(Datei)
  end;

Procedure write_Q(var Datei:PicFileType;x,y:real);
  begin with Qpuf do begin bgx:=x;bgy:=y;end;
    with kopf do begin ebene:=0;farbe:=0;folge:=0;Was:='Q' end;
    satzschreiben(Datei)
  end;

Procedure write_V(var Datei:PicFileType);
  begin
    with kopf do begin ebene:=0;farbe:=0;folge:=0;Was:='V' end;
    satzschreiben(Datei)
  end;

Procedure write_E(var Datei:PicFileType);
  begin
    with kopf do begin ebene:=0;farbe:=0;folge:=0;Was:='E' end;
    satzschreiben(Datei)
  end;
```

*Bild 36*

**Tafel 1 Übersicht**

| Kennung | Daten in | Element |
|---|---|---|
| A | Apuf | Symbol |
| a | aapuf | Kurz-Symbol |
| C | Cpuf | Bemaßungstext |
| d | ddpuf | kurzer Text |
| D | Dpuf | langer Text |
| E | ---- | Ende-Kennung |
| K | Kpuf | Kreis |
| L | Lpuf | Linie |
| m | mmpuf | Dreieckiger Füllblock |
| M | Mpuf | Parallelogram Füllblock |
| N | Npuf | Rechteckiger Füllblock |
| O | Opuf | Punkt |
| Q | Qpuf | Bildmaße |
| R | Rpuf | Faktor für Kurzsymbole |
| V | Vpuf | Versionskennzeichen |
| X | Xpuf | Symbol-Referenzpunkt |

Variablen Kopf.Was, die das rufende Programm einstellen muß. Auf der Grundlage dieser Information ermittelt sie die Länge des zu schreibenden Satzes. Das Schreiben selbst wird wiederum in zwei Etappen durchgeführt. Zuerst wird der Satz-Kopf abgespeichert und dann der entsprechende Rest des Satzes. Lediglich das Versionskennzeichen erfährt eine Sonderbehandlung.

Mit Hilfe dieser beiden Prozeduren ist es nun möglich, Bilddateien sowohl zu lesen als auch zu schreiben. Im Gegensatz zum Lesen, bei dem es der jeweiligen Anwendung überlassen bleibt, welche gelesenen Informationen weiter verwendet werden, muß beim Schreiben stets garantiert werden, daß die abzuspeichernden Datensätze komplett gefüllt sind. Dabei genügt es nicht, der Variablen *Lpuff.1Typ*, also dem Linientyp einmalig einen Wert zuzuweisen. Bei jeder Schreib- oder Leseoperation eines anderen grafischen Elementes kann dieser Wert im Buffer überschrieben werden, da ja alle Buffer-Variablen überlagert sind. Daher ist es sinnvoll, diese Aufgabe speziellen Prozeduren anzuvertrauen, die jeweils einen einzigen Grafik-Satz zum Schreiben anweisen. In diesen Prozeduren können dann auch, wie in den folgenden Beispielen, Standardbelegungen fest programmiert werden.

Als Beispiel soll hier lediglich die Prozedur write_L erläutert werden. Sie schreibt eine Linie in die Bilddatei. Der Prozedur werden als Parameter Anfangs- und Endpunkte der zu speichernden Linie, deren Zeichenebene und Farbe übergeben. Diese Angaben werden korrekt in den Übertragungspfuffern gesetzt. Dabei wird der Linientyp, die Linienstärke und das Folgen-Kennzeichen mit konstanten Werten eingestellt. Wesentlich ist wieder das Setzen des Satzkennzeichens *Kopf.Was*. Somit sind alle zu übertragenden Informationen bereitgestellt und die Prozedur Satzschreiben kann aufgerufen werden. Erst sie schreibt ja die Daten in die Bilddatei.

Zum Schluß sei noch darauf hingewiesen, daß in allen Prozedur-Vereinbarungen enthaltene File-Variable *Datei* natürlich im

```
Program Linien_Selektieren;

{$I PIC.BIB }

var cad,cadneu:PicFileType;

begin
  assign(cad,'Bild.pic');reset(cad);
  assign(cadneu,'Linien.pic');rewrite(cadneu);
  repeat
    satzlesen(cad);
    if kopf.was in ['V','Q','L','E'] then
      satzschreiben(cadneu)
  until Kopf.was = 'E';
  close(cad);
  close(cadneu)
end.
```

*Bild 37*

Hauptprogramm erst zugewiesen werden muß. Weiterhin ist es nicht möglich, auf eine Bilddatei gleichzeitig schreibend und lesend zuzugreifen.
Zur Demonstration der Nutzung dieser Lösung geben wir im Bild 37 ein kleines Programmbeispiel an, welches eine Bilddatei liest und lediglich die Linien dieser Datei auf eine zweite Datei überträgt. Die Sätze mit den Kennungen V, Q oder E, die zu jeder Bilddatei gehören, werden natürlich ebenfalls übernommen. Die in der Include-Anweisung im Programmtext angegebene Datei PIC.BIB enthält die im Bild 36 abgedruckte Lösung.

## Listen

Nachdem wir in den letzten Kapiteln darüber gesprochen haben, wie wir uns Daten beschaffen, wollen wir nun die Verwaltung von Daten im Hauptspeicher betrachten. Es wird dabei vor allem um die effektive Haltung größerer Datenmengen gehen. Die hier vorzustellende Methodik der Daten-Listen ist hauptsächlich für jene Fälle gedacht, in denen der Programmierer beim Entwurf eines Programms nicht weiß, welche konkreten Datenmengen während eines Programmlaufes auftreten werden. Zum Verständnis der folgenden Lösungen ist die Vertrautheit des Lesers mit dem Pointer-Begriff und der Handhabung von dynamischen Variablen Voraussetzung. Diese Beispiele beschäftigen sich durchweg mit Listen- und Baum-Strukturen, zu deren Beschreibung es in der Fachliteratur im allgemeinen feste Sprachregelungen gibt. Der Autor wird sich aber nur lose an diesen Sprachgebrauch halten, da er eine größtmögliche Verständlichkeit speziell für Turbo-Programmierer erreichen möchte.

*Einfach gekoppelte, geordnete Listen*

Die in der ersten Lösung zu besprechenden Listen sollen einfach gekoppelt und geordnet sein. Sie besitzen also genau ein erstes und ein letztes Listenelement. Jedes Listenelement enthält neben dem *Eintrag* eine Variable mit Namen *Naechster*, in dem auf das jeweils nächste Element der Liste verwiesen wird. Dieser Verweis ist ein Pointer. Das letzte Listenelement enthält in diesem Feld den Pointerwert *NIL*. NIL ist ein Pointer der

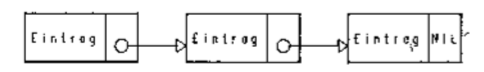



*Bild 38*

nirgendwohin zeigt und hier als Listen-Endekennung verwendet wird. Man kann sich also mit Hilfe der in *Naechster* enthaltenen Pointerwerte innerhalb dieser Liste vorwärts tasten, bis deren Ende erreicht ist. Ein Rückwärtslaufen in der Liste ist dagegen hier nicht möglich, da in den Listenelementen kein Verweis auf den Vorgänger enthalten ist, was natürlich auch möglich wäre. Jedes Listenelement enthält also eine Kopplung zu dem Nachfolgeelement. Dagegen enthält ein solches Element keinen Hinweis auf seinen Vorgänger. Damit ist die Liste nur einfach gekoppelt (siehe Darstellung in Bild 38).
Zum Bearbeiten solcher Listen dienen die drei in Bild 39 gezeigten Funktionen. Das Einfügen eines neuen Listenelementes, das Suchen und das Löschen eines Elements. Dabei soll beim Einfügen für das neue Element Platz im Hauptspeicher reserviert werden und beim Löschen soll dieser Platz wieder freigegeben werden. So wird erreicht, daß die Liste jeweils nur den Platz im Hauptspeicher beansprucht, den sie aktuell benötigt.
Wenn die Elemente der Liste geordnet gespeichert werden sollen, ist ein Ordnungskriterium notwendig. Da diese Ordnungsvorschrift aber für verschiedene Listeninhalte sehr unterschiedlich sein kann, wird für die vorgestellten Listenoperationen die Funktion *kleiner* vorausgesetzt. Selbstverständlich muß auch die Definition des Typs *Listeninhalt* außerhalb dieser Lösung erfolgen, da dieser ja für jede Anwendung unterschiedlich sein wird (Bild 39).
Die erste Komponente der Lösung ist die Prozedur Einfügen. Ihr werden zwei Parameter übergeben. Der erste, mit L bezeichnete, ist ein Pointer, der auf den Anfang der zu erweiternden Liste zeigt. Der zweite Parameter, mit E bezeichnet, ist der Eintrag, der in die Liste aufgenommen werden soll. Zur internen Organisation dieser Prozedur sind zwei weitere Pointervariablen p und q notwendig. Der Pointer p wird zuerst auf den angegebenen Anfang der Liste eingestellt. Die folgende Suchschleife wird genau dann verlassen, wenn die Stelle in der Liste gefunden wurde, an der der Eintrag eingefügt werden soll. Ist die angegebene Liste leer, enthielt der Parameter L also den Wert NIL, erfolgt kein Suchen. Nach dieser Schleife wird der notwendige Speicherplatz für das neue Listenelement mit der Standardprozedur *new* reserviert. Der Pointer p zeigt jetzt auf das neue Element, und es wird mit dem Inhalt des Eintrags E gefüllt. Nun muß dieses erzeugte Listenelement auch noch logisch in die Liste eingebunden werden, das heißt, die Felder *Naechster* des Vorgängers und natürlich auch des neuen Elementes selbst müssen korrekt auf den jeweiligen Nachfolger eingestellt werden. Erst so wird das Element in die Listenordnung eingefügt. Dazu ist aber eine Fallunterscheidung notwendig. Hat der Pointer q nach dem Suchvorgang immer noch den Wert NIL, so muß das Element am Anfang der Liste eingefügt werden, oder aber die Liste existierte noch gar nicht. In diesem Fall wird dem Feld *Naechster* des neuen Elementes der alte Listenanfang (L) übergeben und die Procedur gibt den neuen Listenanfang zurück. Im anderen Fall erfolgt ein echtes Einfügen in die alte Liste. Dazu muß der Vorgänger, auf den jetzt die Variable q zeigt, auf das neue Element zeigen, und dieses muß auf das folgende Element zeigen, auf das zuvor der Vorgänger zeigte. Eine Sonderbetrachtung für das Listen-Ende ist dabei nicht notwendig.
Der Prozedur Löschen werden zwei Pointer als Parameter übergeben. Der erste Pointer (L), ist wiederum ein Zeiger auf den Anfang der zu behandelnden Liste. Der zweite Parameter, der Pointer P, zeigt auf das zu löschende Element der Liste. Dabei muß gesagt werden, daß diese Prozedur nicht prüft, ob P wirklich auf ein Listenelement zeigt. Ist dies nicht der Fall, kann das Schaden anrichten. Zur internen Arbeit der Prozedur ist eine weitere Pointervariable (q) notwendig. In der Prozedur müssen drei Fälle unterschieden werden. Im ersten Fall zeigt das zu löschende Element noch auf ein weiteres Listenelement. Dann ist das Löschen auf eine sehr einfache Art und Weise möglich. Das zu

```
(* G_Liste.INC......Operationen zu geordneten Listen *)

type Liste          = ^ListenEintrag;
     ListenEintrag = record
                       Eintrag    : Listeninhalt;
                       Naechster  : Liste
                     end;
procedure Einfuegen (var L:Liste;E:ListenInhalt);
  var p,q:Liste;
  begin
    q:=NIL;p:=L;
    while p<>NIL do with p^ do
      if kleiner(Eintrag,E)
        then begin q:=p;p:=naechster end
        else p:=NIL;
    new(p);p^.Eintrag:=E;
    if q=NIL
      then begin p^.Naechster:=L;L:=p end
      else begin p^.Naechster:=q^.Naechster;q^.Naechster:=p end:
  end;

procedure Loeschen (var L:Liste;P:Liste);
  var q:Liste;
  begin
    q:=P^.Naechster;
    if q<>NIL
      then begin P^:=q^;dispose(q) end
      else
        if P=L
          then begin L:=NIL;dispose(p) end
          else
            begin q:=L;
              while q^.Naechster<>P do q:=q^.Naechster;
              q^.Naechster:=P^.Naechster;dispose(P)
            end
  end;

procedure Suchen(L:Liste;E:ListenInhalt;var P:Liste);
  var gefunden:boolean;
  begin
    P:=L;gefunden:=false;
    while ( P<>NIL ) and ( not gefunden ) do
      begin
        gefunden:= not( kleiner(P^.Eintrag,E) or kleiner(E,P^.Eintrag) );
        if not gefunden then P:=P^.Naechster
      end;
  end;
```

*Bild 39*

löschende Listenelement wird mit seinem Nachfolger überschrieben, und der alte Platz des Nachfolgers wird freigegeben. Jetzt zeigt der Vorgänger des gelöschten Elements auf dessen Nachfolger, da dieser ja den Platz des gelöschten Elementes eingenommen hat, und dieser zeigt weiterhin auf dessen Nachfolger. Die logische Folge der Elemente wurde also erhalten. Der zweite Fall, der eintreten kann, ist gegeben, wenn das zu löschende Element einer Liste das einzige in dieser Liste ist. Tritt dies ein, wird dem rufenden Programm über die Variable L mitgeteilt, daß die Liste nun nicht mehr existiert, und der Platz des letzten Elementes dieser Liste wird freigegeben. Am schwierigsten gestaltet sich der Löschvorgang, wenn das zu löschende Element das letzte einer Liste ist, die jedoch später noch weitere Elemente aufnehmen soll. Hierbei ist es notwendig, den Vorgänger des zu löschenden Elementes zu ermitteln, da ja in ihm der Verweis auf das nächste Element gelöscht werden muß. Die Ermittlung des Vorgängers ist bei der gewählten Listenorganisation nur möglich, indem wir uns vom Anfang der Liste bis an die interessierende Stelle vortasten. Nachdem im so gefundenen Vorgänger der Verweis auf das Folgeelement mit NIL überschrieben wurde, kann der belegte Platz des gelöschten Elementes freigegeben werden.
Die dritte Listenoperation wurde zum Suchen eines Listenelementes entwickelt. Ihr wird mit Hilfe des Parameters L wiederum ein Zeiger auf den Anfang der zu durchsuchenden Liste übergeben. Der Parameter E enthält den Eintrag, der gesucht werden soll, und über den Parameter P gibt sie den Zeigerwert zurück, der auf das gefundene Listenelement zeigt. Enthält die Liste das zu suchende Element nicht, wird über diesen Parameter der Wert NIL zurückgegeben. Die Prozedur enthält im wesentlichen nur die Suchschleife selbst. Der in ihr enthaltene Test sieht auf den ersten Blick etwas kompliziert aus. Er nutzt die vorausgesetzte Funktion *kleiner* und prüft die Identität.
Zur Illustration zeigt Ihnen Bild 40 ein kleines Programm. In ihm werden als Listenelemente einfache Integer-Zahlen vereinbart, und für diese Zahlen wird auch die Boolesche Funktion kleiner aufgestellt. Zuerst wird der Pointer-Variablen L, die jeweils den Listenanfang markieren soll, der Wert NIL zugewiesen. Danach wird die Liste mit 20 zufälligen Werten gefüllt. Diese Werte werden geordnet in der Liste abgespeichert. Davon überzeugt uns die dann folgende Schleife, die den entstandenen Listeninhalt anzeigt. Nun werden alle Zahlen mit dem Wert 1 aus der Liste entfernt und der verbleibende Rest der Liste wird wiederum ausgegeben. Bei allen diesen Aktionen dient der Pointer P als Zeiger zu Listenelementen, die jeweils adressiert werden sollen. Die erste und letzte Anweisung im Programm garantieren, daß nach dem Programmlauf der Speicherplatz aller dynamischen Variablen wieder freigegeben ist.
Zusammenfassend: Der für die Verwaltung der Liste erforderliche Speicherplatz ist mit einem Pointer in jedem Listenelement auf ein Minimum beschränkt. Dies hat zur Folge, daß die aufgebauten Listen sofort nur vorwärts, nicht aber auch rückwärts durchgemustert werden können. Anders gesagt: Es ist nicht trivial, zu einem Element den Vorgänger zu ermitteln. Weiterhin sind sowohl in der Komponente Einfügen wie auch in der Komponente Löschen Schleifen enthalten, die natürlich Zeit benötigen. Will man die Elemente der Liste wirklich geordnet haben, ist dies nicht zu ändern. Handelt es sich aber um Listeninhalte, die gar nicht geordnet benötigt werden, sollte man die folgende Lösung benutzen.

*Doppelt gekoppelte, ungeordnete Listen*
Als eine zweite Variante hier nun eine Lösung für die Verwaltung von doppelt gekoppelten, ungeordneten Listen. Damit wird das Suchen sowohl vorwärts als auch rückwärts möglich sein. Dies erfordert aber, in jedem Listenelement zwei Verweise auf die Nachbarelemente in der Liste zu führen. Ein Verweis zeigt dabei auf das jeweils vorhergehende Element, und der zweite Verweis zeigt auf das nachfolgende Element. Die Liste ist somit *doppelt gekoppelt*. Im zweiten Element einer Liste sind also folgende Verweise enthalten: *Vorgaenger* zeigt auf das erste Element und *Nachfolger* zeigt auf das dritte Element der Liste. Das erste Element einer Liste enthält daher im Feld Vorgaenger NIL und das letzte Element einer Liste enthält im Feld Nachfolger NIL (Bild 41). Somit ist ein Durchlaufen der Liste in beiden Richtungen leicht möglich. Da auch auf ein Ordnungskriterium für die Listeninhalte verzichtet wurde, sind die Prozeduren zur Arbeit mit der Liste sehr einfach. Auf Schleifen konnte außer beim Suchen selbst verzichtet werden (Bild 42).
Die Prozedur *Dazu* ermöglicht das Einfügen eines neuen Elementes mit dem Eintrag E in die Liste, auf deren erstes Element der Pointer L zeigt. Da wir ja keine Ordnung in der Liste benötigen, kann das Einfügen des neuen Elementes an beliebiger Stelle der Liste erfolgen. Am einfachsten ist das Einfügen am Anfang der Liste, wie Sie es hier sehen. Zuerst wird für die dynamische Variable, die das einzufügende Element aufnehmen wird, Platz im Hauptspeicher reserviert und sie dann gefüllt. Da es das neue erste Element der Liste sein wird, wird dem Feld Vorgänger der Wert NIL zugewiesen. Das Feld Nachfolger dagegen zeigt auf das alte erste Element der Liste, wobei es unerheblich ist, ob diese Liste bereits existierte. Das alte erste Element der Liste muß nun natürlich in seinem Feld Vorgänger auf das neue erste Element der Liste verweisen. Diese Zuweisung erfolgt nur, wenn die Liste zuvor bereits Elemente enthielt. Die Prozedur Dazu übergibt dem rufenden Programm über den Parameter L den Zeiger auf den neuen Anfang der Liste.

*wird fortgesetzt*

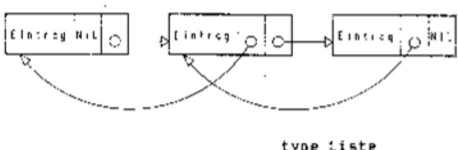



**Bild 41**

```
Program Integer_Sortieren;

type Listeninhalt = integer;

function kleiner(x,y:integer):boolean;
  begin
    if x<y then kleiner:=true else kleiner:=false
  end;

{$I G_Liste.INC }

var i:integer;
    Heaptop,L,P:Liste;
begin
  Mark(heaptop);
  L:=NIL;
  for i:=1 to 20 do Einfuegen(L,random(10));
  P:=L;
  while P<>NIL do
    begin write(P^.Eintrag:4); P:=P^.Naechster end;
  Suchen(L,1,P);
  while P<>NIL do
    begin Loeschen(L,P); Suchen(L,1,P) end;
  P:=L;
  while P<>NIL do
    begin write(P^.Eintrag:4); P:=P^.Naechster end;
  release(heaptop);
end.
```

*Bild 40*

```
type Liste          = ^ListenEintrag;
     ListenEintrag = record
                       Eintrag    : Listeninhalt;
                       Vorgaenger,
                       Nachfolger  : Liste
                     end;

procedure Dazu (var L:Liste;E:ListenInhalt);
  var p:Liste;
  begin
    new(p);p^.Eintrag:=E;p^.Vorgaenger:=NIL;p^.Nachfolger:=L;
    if L<>NIL then L^.Vorgaenger:=p;
    L:=p;
  end;

procedure Weg (var L:Liste;P:Liste);
  var V,N:Liste;
  begin
    N:=P^.Nachfolger;V:=P^.Vorgaenger;
    if N<>NIL then N^.Vorgaenger:=V;
    if V<>NIL then V^.Nachfolger:=N else L:=N;
    dispose(P);
  end;

procedure Finde(L:Liste;E:ListenInhalt;var P:Liste);
  begin
    P:=L;
    while (P<>NIL) and (not(P^.Eintrag = E)) do P:=P^.Nachfolger;
  end;

Procedure Ende(L:Liste;var P:Liste);
  begin
    P:=L;
    while P^.Nachfolger<>NIL do P:=P^.Nachfolger;
  end;
```

*Bild 42*

# Turbo-Pascal-Praxis

Teil 5

**Manfred Zander, Dresden**

*Im Teil 4 unseres Kurses haben wir uns dem Lesen, Bearbeiten und Schreiben von Dateien anderer Programme gewidmet und Ihnen die ersten Prozeduren für geordnete und ungeordnete Listen vorgestellt. Im Bild 42 (MP 12/89, Seite 370) sind die drei Prozeduren Dazu, Weg und Finde als Listing dargestellt. Wir beendeten den Teil 4 mit der Erläuterung der Prozedur Dazu und setzen nun fort mit den Prozeduren Weg und Finde, bevor wir dann die Arbeit mit Daten in Baumstruktur behandeln.*

Das Löschen eines Listenelementes wird mit der Prozedur *Weg* (Bild 42, MP 12/89, S. 370) angewiesen. Der Parameter L zeigt auf die Liste, in der ein Element gelöscht werden soll, der Parameter P zeigt auf das zu löschende Element. Eine Prüfung, ob der Zeiger P wirklich auf ein Element der Liste L zeigt, erfolgt jedoch nicht. Um diese Prozedur übersichtlicher zu gestalten, wurden zwei lokale Pointer vereinbart. Der Pointer N erhält den Zeiger auf den Nachfolger, der Pointer V erhält den Zeiger auf den Vorgänger, des zu löschenden Elementes zugewiesen. Existiert ein Nachfolger, so muß er nach dem Löschen in seinem Feld Vorgänger auf den Vorgänger des zu löschenden Elementes zeigen. Existiert ein Vorgänger, so muß dieser in seinem Feld Nachfolger nach dem Löschen auf den Nachfolger des zu löschenden Elementes zeigen. Wenn das zu löschende Element der Liste keinen Vorgänger besitzt, also das erste der Liste ist, wird dem rufenden Programm über den Parameter L der neue Anfang der Liste mitgeteilt. Nun kann der Speicherplatz, den das aus der Liste gestrichene Element beanspruchte, wieder freigegeben werden.

Zum Suchen eines Listenelementes wurde die Prozedur *Finde* erstellt. Sie hat drei Aufrufparameter. Mit dem Parameter L wird ihr der Zeiger auf das erste Element der zu durchsuchenden Liste übergeben. Im Parameter E steht der Listeninhalt, der gesucht werden soll. Über den Parameter P gibt die Prozedur den Zeiger auf das gefundene Element zurück. Wurde das entsprechende Element nicht gefunden, übergibt die Prozedur den Wert NIL an das rufende Programm. Das Suchen in der Liste erfolgt vom ersten Element an bis der gesuchte Inhalt gefunden wurde, oder aber das Ende der Liste erreicht wurde. Da bei dem gewählten Aufbau der Liste ein Bearbeiten sowohl vom Anfang als auch vom Ende her möglich ist, wird oft die Aufgabe bestehen, das letzte Element zu ermitteln. Dies wird von der Prozedur *Ende* erledigt. Ihr wird über den Parameter L der Zeiger auf ein beliebiges Listenelement übergeben. Mit Hilfe dieses Zeigers läuft sie bis zum letzten Element der Liste und übergibt im Parameter P den Zeiger auf dieses letzte Element der Liste. Die Laufanweisung entspricht der in der Prozedur *Finde* ohne Test auf Identität.

Zur Demonstration der Arbeit mit diesen Prozeduren folgt in Bild 43 ein kleines Programm, welches die beschriebenen Listenoperationen nutzt.

Als Listeninhalt wird wieder der Typ *Integer*

```
Program Trenne_Gerade_Ungerade;

type Listeninhalt = integer;

{$I Liste.INC }

var i,e:integer;
    Heaptop,Lgerade,Lungerade,P:liste;
begin
  Mark(heaptop);
  Lgerade:=NIL;Lungerade:=NIL;
  for i:=1 to 20+random(20) do
    begin
      e:=random(10);
      if (e mod 2 = 0) then Dazu(Lgerade,e)
                       else Dazu(Lungerade,e)
    end;
  Finde(Lgerade,0,P);
  while P<>NIL do
    begin Weg(Lgerade,P); Finde(Lgerade,0,P) end;
  P:=Lgerade;writeln('Die geraden:');
  while P<>NIL do
    begin write(P^.Eintrag:4); P:=P^.Nachfolger end;
  P:=Lungerade;writeln;writeln('Die ungeraden:');
  while P<>NIL do
    begin write(P^.Eintrag:4); P:=P^.Nachfolger end;
  Ende(Lgerade,P);
  writeln;writeln('die geraden rueckwaerts:');
  while P<>NIL do
    begin write(P^.Eintrag:4); P:=P^.Vorgaenger end;
  release(heaptop);
end.
```

***Bild 43** Demonstrationsprogramm für ungeordnete Listen*

vereinbart. Die vereinbarten Pointer-Variablen *Lgerade* und *Lungerade* werden als Zeiger zu zwei unterschiedlichen Listen genutzt, der Pointer P wird als Laufzeiger zu Elementen beider Listen eingesetzt. Zuerst wird den beiden Zeigern Lgerade und Lungerade der Wert NIL zugewiesen; die beiden Listen sind also noch leer. In der folgenden Laufanweisung mit zufälliger Laufgrenze werden Zufallszahlen ermittelt. Die geraden Zufallszahlen werden in der Liste *Lgerade*, die ungeraden Zahlen werden in der Liste *Lungerade* gespeichert. Danach werden alle Elemente der geraden Liste, die den Wert Null enthalten, gelöscht. Im Anschluß daran werden beide Listen auf dem Bildschirm angezeigt. Dabei wird mit dem jeweils ersten Listenelement begonnen. Zur Demonstration der Rückwärtsverarbeitung wird danach der Inhalt der geraden Liste nochmals vom letzten Element an ausgegeben. Die erste und letzte Anweisung im Programm garantieren wiederum, daß alle Reservierungen für dynamische Variablen im Hauptspeicher nach dem Programmlauf wieder freigegeben sind.

Zu den Vor- und Nachteilen dieser Listenorganisation ist folgendes zu sagen. Durch die Einführung eines zweiten Zeigers in den Listenelementen wurden zwei entgegengesetzte Effekte erreicht: Zum einen steigt der Speicherbedarf je Listenelement, zum anderen ist damit die Arbeit mit der Liste wesentlich vereinfacht worden. Durch diese Vereinfachung entstehen vor allem Vorteile bei der Nutzung bzw. Auswertung der Listen, und die Geschwindigkeit der Pointer-Adressierung wird bei dieser Lösung besonders deutlich.

Die beiden vorgestellten Listenstrukturen sind keineswegs die einzig möglichen. Denkbar ist zum Beispiel, bei den geordneten Listen ebenfalls einen Vorgänger-Verweis in den Elementen zu führen, oder aber diesen Verweis bei den ungeordneten Listen wegzulassen. Für welche spezielle Struktur man sich entscheidet, hängt letztendlich vom konkreten Problem ab. Die speziell zu programmierende Aufgabe wird auch entscheiden, welche Vereinfachungen möglich und welche weiteren Listenoperationen erforderlich sind. Weitere Operationen könnten beispielsweise das Einfügen an einer bestimmten Listenoperation, oder auch das Tauschen von zwei Listenelementen sein. Die Programmierung solcher Lösungen ist mit der vorgestellten Methodik leicht möglich.

## Bäume

In vielen Aufgabenstellungen sind Daten zu verwalten, die untereinander wohl definierte *Unterstellungsverhältnisse* haben. Als Beispiel kann die Organisationsstruktur eines Betriebes dienen. Ein Betrieb hat genau einen Kopf, den Betriebsdirektor. Diesem Kopf sind die jeweiligen Fachdirektoren untergeordnet. Den Fachdirektoren unterstehen jeweils mehrere Hauptabteilungsleiter, diesen die Abteilungsleiter und so weiter. Bei einer Liste gibt es genau ein erstes und ein letztes Element; sie ist *linear* aufgebaut und damit zur Beschreibung dieser Struktur nicht geeignet. In einem Baum dagegen existiert ebenfalls genau ein erstes Element, es ist aber kein letztes definiert. Jedes Element (oder Knoten) eines Baumes hat genau einen Vorgänger, aber eine beliebige Anzahl von Nachfolgern. So untersteht ein Hauptabteilungsleiter genau einem Direktor, selbst aber unterstehen ihm mehrere Abteilungsleiter. Bäume sind also nicht linear, sondern *verzweigt* aufgebaut. Ein Spezialfall dieser Problematik ist der binäre Baum, den wir zuerst behandeln wollen.

*Der binäre Baum*

Ein binärer Baum hat genau einen Kopfknoten, der selbst keinen Vorgänger hat. In diesem *Baum-Kopf* sind, wie in jedem anderen Knoten des Baumes auch, jeweils zwei Verweise enthalten, die auf weitere Knoten des Baumes zeigen können. Als Beispiel bietet sich der Stammbaum einer Person an. Der Kopf des entsprechenden Baumes ist die Person selbst. In diesem Kopf gibt es genau zwei Verweise, die auf die Mutter und den Vater zeigen. Die *Knoten* der Mutter und des Vaters wiederum zeigen auf jeweils zwei weitere Knoten, auf die Großeltern. Diese Verweise können, wenn notwendig, immer weiter gefächert werden. Wenn wir den Kopf ei-

nes solchen Baumes als erste Ebene bezeichnen, hat die zweite Ebene, die der Eltern, zwei Einträge, die dritte Ebene kann maximal vier Einträge aufnehmen, die vierte acht und so weiter. Um in einer solchen Struktur von einem beliebigen Knoten aus wieder zum Kopf zu finden, wird in jedem Knoten ein Zeigerwert auf den jeweils übergeordneten gestellt. Damit ergibt sich für ein Knotenelement des binären Baumes folgender Aufbau: Der Eintrag selbst, ein Zeiger zum übergeordneten Knoten, im Beispiel **v**

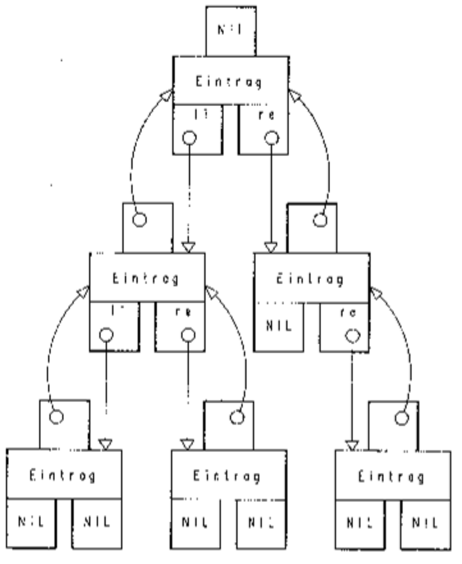



*Bild 44 Der binäre Baum*

bezeichnet, sowie zwei weitere Zeiger zu den jeweils untergeordneten Knoten, bezeichnet mit links und rechts (Bild 44).
Zur Arbeit mit einer solchen Baumstruktur dienen die in Bild 45 gezeigten Prozeduren. Die Prozedur *Kopfset* organisiert den Beginn des Baumaufbaus; sie reserviert den Speicherplatz für den Kopfknoten und füllt dessen Eintrag. Dann setzt sie alle drei Verweise auf NIL, da dieser Knoten ja noch auf keinen weiteren Knoten zeigt. Über den Parameter L gibt sie dem rufenden Programm den Zeiger auf diesen Kopf des Baumes zurück. Der weitere Aufbau des Baumes erfolgt dann mit der Prozedur *Anfügen*, die die Existenz von mindestens einem Baumknoten voraussetzt. Dieser Prozedur werden in der vorgestellten Realisierung insgesamt fünf Parameter übergeben. Im ersten Parameter R erfolgt die Spezifizierung, ob der neue Knoten rechts oder links angebunden werden soll. Der Zeiger L zeigt auf den jeweiligen Baum, der erweitert werden soll. Die Prozedur hat den Auftrag, einen Knoten mit dem Eintrag E an den Knoten anzubinden, der den Eintrag AE enthält. Über den fünften Parameter P wird nach erfolgreicher Arbeit der Zeiger auf den neuen Knoten zurückgegeben. Hat der Parameter P nach dem Beenden der Prozedur den Wert NIL, konnte der Auftrag nicht erfüllt werden. Das Vorgehen der Prozedur Anfügen kann folgendermaßen beschrieben werden. Zuerst ermittelt sie den Zeigerwert zu den Knoten, an den angefügt werden soll. Dies wird mit Hilfe der Prozedur *Finde* realisiert. Nach Aufruf dieser Suchprozedur enthält die Pointervariable *Tem* den Zeigerwert auf den Knoten mit dem Eintrag AE. Ist dieser Zeiger gleich NIL, so existiert kein Knoten AE im Baum; ein Anfügen ist also nicht möglich. Ist der Knoten, an den angefügt werden soll, vorhanden, wird ermittelt, ob der gewünschte Verweis links oder rechts den Wert NIL besitzt. Dieser Test ist notwendig, da es beispielsweise nicht möglich ist, an einen Knoten zwei rechte Knoten anzubinden. Ist auch dieser Test positiv verlaufen, wird das Anfügen vorgenommen. Dazu wird für den neuen Zweig Platz reserviert, mit dem Eintrag gefüllt wird. Die notwendigen Referenzen in ihm wie auch im übergeordneten Knoten werden ordnungsgemäß gesetzt. Der neue Knoten weist mit **v** auf den alten Knoten zurück und mit **re** und **ll** auf NIL. Der alte Knoten weist, wie mit **R** entschieden, entweder mit re oder aber mit li auf den neuen Knoten.
Diese hier vorgestellte Prozedur ist aber nur ein Beispiel. Das jeweils zu lösende Problem wird bestimmen, ob die Tests notwendig sind, und wie viele Parameter wirklich benötigt werden. Wenn von außen garantiert ist, daß alle Konventionen eingehalten werden, ist zum Beispiel die Prozedure *Anf* völlig ausreichend. Ihr wird lediglich der Eintrag E des neuen Knotens und ein Zeiger P auf den Knoten übergeben, an den angefügt werden soll. Der dritte Parameter R spezifiziert wieder in die Richtung, in die weiter angefügt werden soll.
Als letzte soll die Prozedur *Finde* beschrieben werden. Sie hat in ihrer Vereinbarung drei Parameter. Mit dem Zeiger L wird ihr mitgeteilt, in welchem Baum sie suchen soll. Der Parameter E enthält den Eintrag, der gesucht wird, und mit Hilfe des Variablen-Parameters P gibt sie den Zeiger auf den gefundenen Knoten zurück. Enthält der Zeiger P nach dem Beenden der Prozedur immer noch den Wert NIL, so wurde der gesuchte Eintrag nicht gefunden. Durch die rekursive Programmierung dieser Suche konnte die Prozedur sehr einfach gehalten werden. Als erstes testet sie ab, ob der Knoten auf den L zeigt, den gesuchten Eintrag besitzt. Ist dies der Fall, wird P der Zeiger L zugewiesen und die Prozedur ist zu Ende. Im anderen Fall ruft sich die Prozedur selbst auf und setzt dabei den Parameter L auf ihren rechten Verweis. Wurde auch rechts nichts gefunden, erfolgt die weitere Suche links. Mit diesen wenigen Zeilen wird gewährleistet, daß sich die Suche über den kompletten aufgebauten Baum erstreckt. Die Prozedur *Finde* ist ein schönes Beispiel für die Vorteile der rekursiven Programmierung. Im nichtrekursiven Stil wäre diese Prozedur bedeutend komplizierter.
Zur Erläuterung der Nutzung von binären Bäumen erfolgt in Bild 46 wieder die Angabe eines kleinen Hauptprogrammes. Als Eintragtyp für die Knoten des Baumes wurden hier Strings der Länge 20 vereinbart. Zuerst wird der Kopf des Baumes mit dem Eintrag *Kind* gesetzt. Danach wird die Verwandtschaft des Kindes aufgenommen. Dieser Aufbau des Baumes, der hier schrittweise erfolgt, kann in echten Anwendungen natürlich anders organisiert werden. So kann das Wachsen des Baumes aber direkt nachvollzogen werden. Zum Aufbau werden sowohl die Prozedur *Anfügen* als auch die einfache Variante *Anf* genutzt, um die Wirkung beider zu illustrieren. Nach dem Aufbau des Baumes folgt dessen Nutzung. Es wird jeweils ein Name gelesen, und dieser Name wird im Baum gesucht. Existiert ein Knoten mit diesem Eintrag, so wird von diesem aus der Weg

```
(* Bin.INC........Prozeduren fuer binaere Baeume *)
($A-)
type Richtung = (rechts,links);
     Knoten   = ^Knotentyp;
     Knotentyp= record
                  Eintrag:Eintragtyp;
                  v,li,re:Knoten
                end;

Procedure Kopfset (var L:Knoten;E:Eintragtyp);
  begin
    new(L);
    with L^ do begin Eintrag:=E;v:=NIL;re:=NIL;li:=NIL end;
  end;

Procedure Finde (L:Knoten;E:Eintragtyp;var P:Knoten);
  begin
    P:=NIL;
    if L^.Eintrag = E then P:=L
    else
      begin
        if L^.re<>NIL then Finde(L^.re,E,P);
        if (P=NIL) and (L^.li<>NIL) then Finde(L^.li,E,P);
      end
  end;

Procedure Anfuegen(R:Richtung;L:Knoten;AE,E:Eintragtyp;var P:Knoten);
  var Tem:Knoten;
  begin
    Finde(L,AE,Tem);P:=NIL;
    if Tem = NIL then writeln('Anfuegeknoten nicht gefunden')
      else begin
        if R=rechts then P:=Tem^.re else P:=Tem^.li;
        if P<>NIL then begin P:=NIL;writeln('Anfuegung besetzt') end
          else begin
            new(P);
            if R=rechts then Tem^.re:=P else Tem^.li:=P;
            with P^ do begin v:=Tem;Eintrag:=E;re:=NIL;li:=NIL end
  end end end;

procedure Anf (P:Knoten;E:Eintragtyp;R:Richtung);
  var Tem:Knoten;
  begin
    new(Tem);
    with Tem^ do begin v:=P;Eintrag:=E;re:=NIL;li:=NIL end;
    if R=rechts then P^.re:=Tem else P^.li:=Tem
  end;
```

*Bild 45 Prozeduren für binäre Bäume*

```
program Stammbaum;

type Eintragtyp = string[20];

{$I Bin.INC }

var Haeptop,L,P: Knoten;    E:Eintragtyp;

begin
  Mark(haeptop);
  kopfset(L,'Kind');
  Anfuegen(links,L,'Kind','Manfred',P);
  Anfuegen(rechts,L,'Kind','Gabi',P);
  Anf(P,'Doris',rechts); Anf(P,'Wilfried',links);
  Finde(L,'Manfred',P);
  Anf(P,'Helga',rechts);    Anf(P,'Heinz',links);
  Anfuegen(rechts,L,'Helga','Auguste',P);
  Anfuegen(rechts,L,'Doris','Marie',P);
  repeat
    write('Ermittelung fuer: ');readln(E);
    Finde(L,E,P);
    while (P^.v <> NIL)and(P<>NIL) do
      begin
        write(' ':20,P^.Eintrag,' ist ');
        if P^.v^.re = P then write('Mutter')
                        else write('Vater');
        writeln(' von: ',P^.v^.Eintrag);
        P:=P^.v
      end
  until E='';
  release(Haeptop)
end.
```

*Bild 46 Programm Stammbaum*

bis zum Kind zurückverfolgt und protokolliert.

Das Bild 47 stellt den im Programm aufgebauten Stammbaum graphisch dar, und Bild 48 zeigt das entstehende Bildschirmaussehen bei der Bedienung des Programms.

*Beliebig verzweigte Bäume*

In binären Bäumen kann ein Knoten jeweils nur auf zwei weitere Knoten verweisen. Dies kann natürlich ein Nachteil sein. In den meisten Aufgabenstellungen wird gefordert, daß ein Knoten auf eine beliebige Anzahl von anderen Knoten verweisen kann. Die im folgenden vorgestellte Methodik zur Lösung dieser Aufgabe kann nur *ein* Beispiel sein und sollte auch nicht dogmatisch übernommen werden, wenn eigene Probleme gelöst werden sollen.

Zuerst wird die zu verwaltende Struktur im Hauptspeicher erläutert werden. Im Gegensatz zu den Listen und den binären Bäumen haben wir es nun mit zwei Typen von dynamischen Variablen zu tun. Der erste Typ, bezeichnet mit Knotentyp, beinhaltet die zu verwaltenden Einträge und zwei unterschiedlich typisierte Pointer. Der erste Pointer, bezeichnet mit *vor*, weist auf den jeweiligen Vorgängerknoten. Der zweite Zeiger in jedem Knoten, bezeichnet mit *nL*, zeigt auf eine lineare Liste, in der alle Verweise auf Nachfolgeknoten gesammelt werden. Die Elemente dieser Verweisliste haben je zwei Felder. Das erste Feld, bezeichnet mit *nach*, verweist jeweils auf einen Nachfolgeknoten, und das Feld *next* dient zur linearen Verkettung dieser Nachfolgerliste. Durch die Speicherung der Nachfolgerverweise in einer Liste ist es nun möglich, zu jedem Knoten eine beliebige Anzahl von Verweisen zu anderen Knoten aufzubauen. Damit hat der aufgebaute Baum wieder genau einen Kopfknoten, aber keinen *letzten* Knoten. Jeder Knoten des Baumes, mit Ausnahme des Kopfes, hat genau einen übergeordneten Vorgänger und eine beliebige Anzahl von untergeordneten Nachfolgern

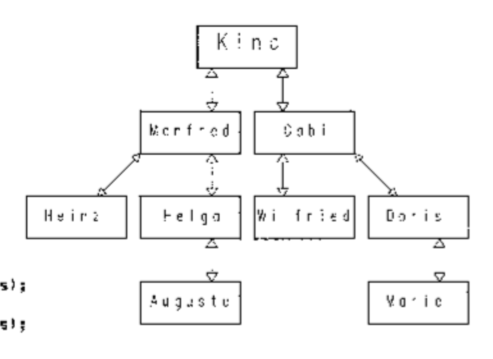
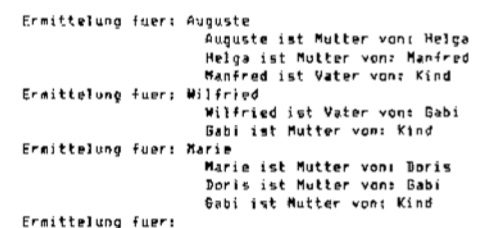



*Bild 47 Darstellung des aufgebauten Baumes*

(Bild 49). Zur Bearbeitung dieser Baumstruktur im Hauptspeicher werden wieder drei Grundprozeduren vorgestellt. *FindeKnoten* soll nach einem Knoten im Baum suchen. Die Prozedur *NeuerKnoten* dient zum Anhängen eines neuen Knotens an einen bestehenden Baum, bzw. zum Aufbauen eines Baumes, und die Prozedur *Zweigab* dient zum Löschen eines Knotens im Baum. Durch diese Löschprozedur werden aber alle dem zu löschenden Knoten untergeordneten Knoten ebenfalls gelöscht werden! Sowohl die Prozedur *FindeKnoten* als auch *Zweigab* sind rekursiv programmiert (Bild 50).



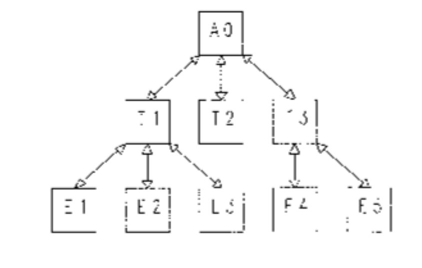
```
Ermittelung fuer: Auguste
                    Auguste ist Mutter von: Helga
                    Helga ist Mutter von: Manfred
                    Manfred ist Vater von: Kind
Ermittelung fuer: Wilfried
                    Wilfried ist Vater von: Gabi
                    Gabi ist Mutter von: Kind
Ermittelung fuer: Marie
                    Marie ist Mutter von: Doris
                    Doris ist Mutter von: Gabi
                    Gabi ist Mutter von: Kind
Ermittelung fuer:
```

*Bild 48 Dialog mit dem Programm Stammbaum*

Der Prozedur *FindeKnoten* wird mit dem Parameter B übergeben, in welchen Baum bzw. Baumteil der Eintrag E gesucht werden soll. Der Zeiger auf den gefundenen Knoten mit diesem Eintrag wird dem rufenden Programm über den Parameter P zurückgegeben. Existiert kein Knoten mit dem Eintrag E im durchsuchten Baum, wird über P der Wert NIL übermittelt. Die Identität zweier Einträge wird in dieser Prozedur mit Hilfe der vorausgesetzten Booleschen Funktion *gleich* getestet. Die Suche im Baum selbst ist wieder rekursiv programmiert. Die Prozedur prüft zuerst, ob der erste Knoten des angegebenen Baumes den Eintrag E enthält. Ist dies der Fall, so wird der Variablen P der Zeiger auf diesen Knoten übergeben. Anderenfalls wird diese Prüfung für alle die Knoten organisiert, auf die in der Nachliste des aktuellen Knotens verwiesen wird. Für jeden dieser untergeordneten Knoten ruft sich die Prozedur selbst auf und realisiert so eine vollständige Durchmusterung aller Baumverzweigungen. Diese Suche wird genau dann beendet, wenn entweder der gesuchte Knoten gefunden wurde oder aber der Baum vollständig durchsucht ist.

Die Prozedur NeuerKnoten dient zum schrittweisen Aufbauen des Baumes im Hauptspeicher. Ihre Aufgabe besteht darin, an den Knoten, auf den der Zeiger *AN* zeigt, einen neuen Knoten anzuhängen und in diesem neuen Knoten den Eintrag E abzulegen. Hat der Parameter *AN* den Wert NIL, so handelt es sich um den neuen Kopfknoten eines neuen Baumes, und der Zeiger auf diesen Kopf wird dem rufenden Programm über AN mitgeteilt. Existiert der Knoten, an den angefügt werden soll dagegen, so muß die logische Verknüpfung des alten und des neuen Knotens realisiert werden. Der Zeiger *vor* des neuen Knotens zeigt auf den alten Knoten. Dies ist mit einer einfachen Zuweisung möglich. Für den Verweis des alten Knotens auf den neuen muß die *Nach*liste des alten Knotens um ein Listenelement erweitert werden. Hat der alte Knoten noch keine *Nach*liste, so muß diese eröffnet werden. Somit ist die eindeutige Bindung des neuen Knotens an den Baum erreicht.

Die Prozedur Zweigab hat vordergründig die Aufgabe, den Knoten, auf den der Parameter P zeigt, zu löschen. Dazu muß aber nicht nur der Knoten selbst freigegeben werden. Gleichzeitig ist es notwendig, den Verweis in der *Nach*liste des übergeordneten Knotens zu löschen. Aber auch das genügt noch nicht. Wenn der zu löschende Knoten selbst noch eine Nachliste besitzt, so wären die Knoten, auf die in der Nachliste verwiesen wird, nicht freigegeben, aber vom Baum logisch abgehängt. Es ist also notwendig, nicht nur den Knoten selbst, sondern auch alle untergeordneten Zweige des Knotens zu löschen. Dies ist wieder nur durch eine rekursive Programmierung der Prozedur zu organisieren. Sie geht folgendermaßen vor: Zuerst ruft sie für alle Knoten, die in der Nachliste des zu lö-



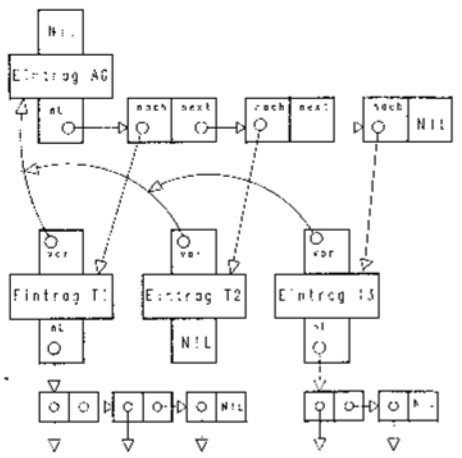




*Bild 49 Der beliebig verzweigte Baum (Prinzip und Beispiel)*

```
(*****************************)
(* Include Baum.INC         *)
(*****************************)

{$A-}   (* Erzeugen rekursiven Codes !!!!! *)
type Knoten     = ^Knotentyp;
     Verweis    = ^Verweistyp;
     Knotentyp  = record Eintrag:Eintragtyp;
                         vor:Knoten;
                         nL :Verweis     end;
     Verweistyp = record nach:Knoten;
                         next:Verweis    end;

Procedure FindeKnoten(Baum:Knoten;E:Eintragtyp;var P:Knoten);
  var V:Verweis;
  begin
    P:=NIL;
    if Baum<> NIL then
      if gleich(Baum^.Eintrag,E) then P:=Baum
      else begin
        V:=Baum^.nL;
        while (V<>NIL)and(P=NIL) do
          begin FindeKnoten(V^.nach,E,P); V:=V^.next
  end end end;
Procedure NeuerAst(var AN:Knoten;E:Eintragtyp);
  var V,L:Verweis;Tem:Knoten;
  begin
    new(Tem);with Tem^ do begin Eintrag:=E;vor:=AN;nL:=NIL end;
    if AN = NIL then AN:=Tem
      else begin
        new(V);with V^ do begin nach:=Tem;next:=NIL end;
        if AN^.nL = NIL then AN^.nL:=V
          else begin
            L:=AN^.nL;
            while L^.next <> NIL do L:=L^.next;
            L^.next:=V
  end   end end;
Procedure Zweigab(Ab:Knoten);
  Var A,V:Verweis;
  begin If Ab<>NIL then begin
    V:=Ab^.nL;
    while V<>NIL do
      begin A:=V^.next;Zweigab(V^.nach);V:=A end;
    if Ab^.vor<>NIL then
      begin
        V:=Ab^.vor^.nL;
        if V^.nach=Ab
          then Ab^.vor^.nL:=V^.next
          else begin
            while V^.nach<>Ab do begin A:=V;V:=V^.next end;
            A^.next:=V^.next end;
        dispose(V) end;
    dispose(Ab)
  end end;
```

*Bild 50 Prozeduren zur Baumbearbeitung*

schenden Knotens enthalten sind, sich selbst auf. Da bei der Löschung eines untergeordneten Knotens auch der Verweis auf diesen Knoten in der Nachliste gelöscht und freigegeben wird, verschwindet die Nachliste somit vollständig. Sind alle untergeordneten Knoten gelöscht, ist die Nachliste also leer; es muß also nur noch der Verweis auf den zu löschenden Knoten im übergeordneten Knoten beseitigt werden. Dies gilt natürlich nicht für den Kopfknoten eines Baumes. Nachdem alle Verweise auf den zu löschenden Knoten beseitigt sind, kann zuletzt auch er selbst noch freigegeben werden.

Mit Hilfe dieser drei Prozeduren zur Baumbehandlung ist es nun möglich, einen Baum im Hauptspeicher auf- und abzubauen sowie einzelne Knoten im Baum aufzufinden. Im weiteren geht es nun darum, den aufgebauten Baum auszuwerten, das heißt aus dem Baum neue Informationen zu gewinnen. Die dabei möglichen Fragestellungen sind zwar sehr stark abhängig vom Charakter der verwalteten Informationen in den Knoten, es lassen sich aber trotzdem einige allgemeingültige Prinzipien und Fragestellungen herausarbeiten. Beispiele für immer wieder auftretende Aufgaben sind die Ermittlung aller übergeordneten oder auch aller untergeordneten Knoten zu einem konkreten Knoten im Baum.

Das Ergebnis einer solchen Aufgabe ist aber nicht ein einzelnes Objekt, sondern meist eine gewisse Anzahl von Knoten. Das Ergebnis ist also eine Liste von Knoten. Nun wäre es aber zu aufwendig, in dieser Liste die kompletten Knoteninhalte nochmals aufzubauen. Es ist völlig ausreichend, nur die Verweise zu den Knoten, die ermittelt worden sind, in der Ergebnisliste zu vermerken. Im Ergebnis solcher Fragestellungen entstehen also Listen, in denen Knotenverweise gesammelt werden.

Diese Listen entsprechen in ihrem Aufbau völlig den *Nach*listen der Baumknoten. Der Unterschied besteht nur darin, daß die Ergebnislisten zwar auf die Knoten des Baumes verweisen, es aber keinen Rückbezug vom Baum auf diese Listen gibt. Die *Nach*listen geben die Struktur des Baumes an, die Ergebnislisten sind eine Sammlung von Knoten. Es besteht also lediglich ein Unterschied in der Nutzung der Listen, kein Unterschied in ihrem physischen Aufbau. Zu einem Baum können also durch verschiedenste Auswertungen beliebig viele Ergebnis- oder Knotenlisten aufgebaut und gelöscht werden, ohne daß der Baum selbst verändert wird.

Es erweist sich wieder als günstig, die Verwaltung der Knotenlisten speziellen Prozeduren zu übertragen. Diese sind in dem Bild 51 aufgelistet.

Von der Struktur her handelt es sich bei den Knotenlisten um einfach verkettete, ungeordnete Listen. Die verwalteten Listeninhalte sind in diesem Fall Pointer auf Knoten. Alle notwendigen Typvereinbarungen sind bereits in den Baumprozeduren vorhanden, und werden hier vorausgesetzt.

Die Prozedur *DazuKnotenListe* erweitert eine Knotenliste. Der erste Parameter stellt einen Zeiger auf die zu erweiternde Liste dar, der zweite Parameter den aufzunehmenden Knotenverweis. Das neue Listenelement wird an den Anfang der zu erweiternden Liste gesetzt, und der neue Listenanfang wird über den ersten Parameter zurückgegeben. Dieses hier angewandte Prinzip entspricht dem in den ungeordneten Listen.

*wird fortgesetzt*

```
(*****************************)
(* Include KnotenLi.INC     *)
(*****************************)

Procedure DazuKnotenListe(var KnotenListe:Verweis;Kn:Knoten);
  var T:Verweis;
  begin new(T);
    with T^ do begin nach:=Kn;next:=KnotenListe end;
    KnotenListe:=T
  end;
Procedure LoeschenKnotenListe(var KnotenListe:Verweis);
  var T:Verweis;
    begin T:=KnotenListe;
      while T<> NIL do
        begin T:=T^.next;dispose(KnotenListe);KnotenListe:=T end
    end;
Procedure DruckenKnotenListe( KnotenListe:Verweis);
    var T:Verweis;
    begin T:=KnotenListe;
      while T<> NIL do
        begin
          Druckeknoten(T^.nach);
          T:=T^.next;
    end;writeln end;
Function Anzahl(L:Verweis):integer;
  var I:integer;
  begin I:=0;while L<>NIL do begin I:=I+1;L:=L^.next end;Anzahl:=I end;
Function Member(L:Verweis;E:Eintragtyp):boolean;
  begin Member:=false;
    while L<>NIL do begin
      if gleich(L^.nach^.Eintrag,E) then Member:=true;
      L:=L^.next
  end end;
```

*Bild 51 Prozeduren für Knoten-Listen*

# Turbo-Pascal-Praxis

**Teil 6**


**Manfred Zander, Dresden**


*Liebe Leserin, lieber Leser,*
*nachdem Sie unserem Kurs Turbo-Pascal-Praxis nun schon über fünf Folgen und fast ein Jahr die Treue gehalten haben, und hoffentlich auch zahlreiche Hinweise, Tips und Kniffe für Ihre tägliche Arbeit entnehmen konnten, lesen Sie heute unseren sechsten und letzten Teil.*
*Wir beenden zunächst die Abhandlungen zu den Baumstrukturen und bieten Ihnen dann neben einem Beispiel für die Programmierung von Eingabemasken das vollständige Listing eines kleinen Texteditors an, den Sie sicher sehr gut in Ihren eigenen Programmen nutzen können.*

Das Löschen eines einzelnen Elementes einer Knotenliste ist nicht vorgesehen, da eine Liste später immer als eine Einheit, als komplettes Ergebnis einer Opteration betrachtet werden soll. Demgegenüber ist es natürlich notwendig, das Löschen von kompletten Listen vorzusehen. Diese Aufgabe wird von der Prozedur *LöschenKnotenListe* wahrgenommen (vgl. Bild 51, MP 3/90, S. 82). Dazu wird ihr der Zeiger auf die zu löschende Liste übergeben. Nachdem die Prozedur die Elemente der übergebenen Knotenliste vollständig freigegeben hat, wird über diesen Zeiger der Wert NIL zurückgegeben, die Liste existiert nicht mehr.
Zur Auswertung aufgebauter Knotenlisten sind zwei Funktionen und eine Prozedur erforderlich. Die Prozedur *DruckeKnotenListe* zeigt den Inhalt einer Liste auf dem Bildschirm an. Sie setzt ihrerseits eine Prozedur mit Namen *DruckeKnoten* voraus, da für verschiedene Knoteneintrag-Typen die Bildschirmanzeige unterschiedlich sein kann. Die Prozedur *DruckeKnotenListe* ist daher darauf beschränkt, alle in der Liste gesammelten Knoten für die Anzeige bereitzustellen.
Mit der Funktion *Anzahl* ist es möglich, die Elementezahl einer übergebenen Knotenliste zu ermitteln. Sie durchläuft die Liste von Anfang bis zum Ende und zählt dabei die einzelnen Knotenverweise zusammen. Die Liste selbst wird dabei nicht verändert; zurückgegeben wird die Anzahl als Integerwert.
Zum Test, ob ein Knoten in einer Knotenliste enthalten ist, wurde die Boolesche Funktion *Member* erstellt, die die Funktion *gleich* nutzt (s. Prozedur *FindeKnoten*).
Nachdem die Verwaltung der Ergebnis-Knotenlisten den vorgestellten Prozeduren anvertraut wurde, kann zur Baumanalyse übergegangen werden. Alle im Bild 52 vorgestellten Analyseprozeduren übergeben als Ergebnis eine Knotenliste, in der die jeweils ermittelten Knoten gesammelt sind. Zur Erläuterung der einzelnen Analysen soll wieder die Organisationsstruktur eines Betriebes dienen, wobei in jedem Knoten jeweils eine Person gespeichert sein soll.
Die erste abgedruckte Analyseprozedur mit dem Namen *ZweiginListe* sammelt alle einem Knoten untergeordneten Knoten in einer Liste, einschließlich des Knotens selbst. Auf einen Abteilungsleiter angewandt, enthält die Knotenliste beispielsweise alle Gruppenleiter und alle Mitarbeiter der Abteilung. Für den Betriebsdirektor enthält die entstehende Liste alle Betriebsangehörigen. Die wiederum rekursive Prozedur geht dabei folgendermaßen vor:
Zuerst wird der Knoten, auf den der Parameter B zeigt, in die zu erstellende Liste aufgenommen. Dies wäre in unserem Fall der Abteilungsleiter. Anschließend ruft sich die Prozedur für alle in der *Nach*liste gespeicherten Knoten selbst auf, also für alle Gruppenleiter. Bei der Bearbeitung der Gruppenleiter trägt die Prozedur diese wieder in die Liste ein und ruft sich dann weiter für die Mitarbeiter auf. Durch dieses rekursive Prinzip ist gewährleistet, daß alle an einen Knoten – hier den Abteilungsleiter – anschließenden Zweige erreicht werden. In der entstehenden Liste sind dann alle Personen der Abteilung vom Abteilungsleiter selbst bis zum einzelnen Mitarbeiter enthalten. Die Reihenfolge der Listeneintragungen ist dabei willkürlich und für die weitere Auswertung kaum informativ. Die Prozedur *ZweiginListe* übergibt über den Parameter *Liste* den Zeiger auf die erstellte Knotenliste, die nun zur weiteren Verarbeitung, beispielsweise zur Ausgabe, bereitsteht. Die nächste Analyse-Prozedur mit dem Namen *UnterinListe* sammelt dagegen nur die einem speziellen Knoten B (zum Beispiel dem Abteilungsleiter) direkt untergeordneten Knoten (die Gruppenleiter) in einer Liste. Die dabei entstehende Knotenliste entspricht völlig der *Nach*liste *nL* des Abteilungsleiters. Im Unterschied zu der im Baum verankerten *Nach*liste kann die entstandene Knotenliste aber beliebig geändert werden, ohne den Baum zu *beschädigen*. Die Prozedur *UnterinListe* übergibt über den Variablenparameter *Liste* wieder einen Zeiger auf die erarbeitete Knotenliste.
Die Prozedur *AlleEndinListe* ist rekursiv aufgebaut und ähnelt der Prozedur *ZweiginListe*. Die entstehende Liste enthält aber nur *die* Mitarbeiter, die selbst keine Unterstellten mehr haben. Wird der Prozedur über den Parameter B der Zeiger auf einen Abteilungsleiter übergeben, so werden in der Knotenliste alle Mitarbeiter der Abteilung ohne Leitungsfunktion gesammelt. Zeigt B auf den Betriebsdirektor, so werden alle Mitarbeiter des Betriebes, die keine Leitungsfunktion ausüben, in die Liste aufgenommen. Durch das rekursive Vorgehen der Prozedur ist wiederum gesichert, daß alle Verzweigungen des Baumes berücksichtigt werden.
Die letzte vorgestellte Analyse erhielt den Namen *RückinListe*. Sie ermittelt den Weg von einem übergebenen Knoten P zurück zum Kopfknoten des Baumes. Dabei speichert sie alle Knoten, die auf diesem Weg liegen, in einer Knotenliste ab. Zeigte der Parameter P zum Beispiel auf einen Gruppenleiter, so enthält die entstehende Knotenliste lediglich seine Leiter bis hin zum Betriebsdirektor. Auch diese Prozedur übergibt dem rufenden Programm einen Zeiger auf die entstandene Ergebnisliste.
Weitere Analysen könnten beispielsweise folgende Aufträge erledigen: Ermittlung aller Leiter eines Betriebes; Ermittlung aller Leiter der gleichen Leitungsebene; beides für den gesamten Betrieb oder auf Bereiche eingegrenzt; Ermittlung des korrekten Dienstweges zwischen zwei vorgegebenen Personen in der Struktur usw. Solche und weitere Aufträge konkreter Anwendungen sind im Rahmen der vorgestellten Methodik leicht zu erarbeiten.
Zur Demonstration der Arbeit mit einem beliebig verzweigten Baum und den vorgestellten Analysenprozeduren dient wieder ein kleines Programm (Bild 53). In diesem Beispiel werden die in den Bildern 50 bis 52 abgedruckten Teilkomplexe über Include-Anweisungen eingefügt. Die Reihenfolge der Einfügungen darf dabei nicht verändert werden. Auch die Stellung der Funktion *gleich* und der Prozedur *DruckeKnoten* im Verhältnis zu den Einfügungen ist nur so möglich. Der Anweisungsteil dieser beiden Unterprogramme dagegen wird von Lösung zu Lösung unterschiedlich sein, und ist speziell

```
(******************************)
(* Include Baumanal.INC       *)
(******************************)

{$A-}   (* Erzeugen rekursiven Codes !!!!! *)
procedure ZweiginListe(B:Knoten;var Liste:Verweis);
  var V:Verweis;                    (* Sammeln aller untergeordnetten *)
  begin DazuKnotenListe(Liste,B);   (* Knoten in einer Liste          *)
    V:=B^.nL;
    while V<>NIL do
      begin ZweiginListe(V^.nach,Liste);V:=V^.next
  end end;
procedure UnterinListe(B:Knoten;var Liste:Verweis);
  var V:Verweis;                    (* Sammeln aller direkt unterge-  *)
  begin V:=B^.nL;                   (* ordnetten Knoten in einer Liste *)
    while V<>NIL do
      begin DazuKnotenListe(Liste,V^.nach);V:=V^.next end
  end;
procedure AlleendinListe(B:Knoten;var Liste:Verweis);
  var V:Verweis;                    (* Sammeln aller End-Knoten in    *)
  begin                             (* in einer Liste                 *)
    if B^.nL=NIL then DazuKnotenListe(Liste,B)
    else begin V:=B^.nL;
      while V<>NIL do begin AlleendinListe(V^.nach,Liste);V:=V^.next
  end end end;
procedure RueckinListe(P:Knoten;var Liste:verweis);
  begin Liste:=NIL;                 (* Sammeln aller Knoten von P auf-*)
    while P<>NIL do                 (* waerts in einer Liste          *)
      begin DazuKnotenListe(Liste,P);P:=P^.vor
  end end;
```

*Bild 52 Prozeduren zur Baumanalyse*

```
program Baumanalyse;

type Eintragtyp = record ident:string[20];zahl:integer end;

Function Gleich(E1,E2:Eintragtyp):boolean;
  begin if E1.ident=E2.ident then gleich:=true else gleich:=false end;

(*$I Baum.INC  *)   (* Operationen fuer beliebig verzw.Baeume *)

Procedure DruckeKnoten(P:Knoten);
  begin write(P^.Eintrag.ident,' ':20-length(P^.Eintrag.ident)) end;

(*$I KnotenLi.INC *)        (* Operationen fuer Knoten-Listen *)
(*$I Baumanal.INC *)        (* Operationen fuer Baum-Analyse  *)

var   Haeptop,P,B:Knoten;
      List:Verweis;E:Eintragtyp;
      wahl:char;

begin (* Hauptprogramm *)
  mark(Haeptop);B:=NIL;List:=NIL;
  write('Eingabe Kopfknoten:');readln(E.ident);
  NeuerAst(B,E);              (* Setzen Kopfknoten *)
  repeat
    write('Anfuegen an:');readln(E.ident);FindeKnoten(B,E,P);
    if E.ident<>'Ende' then
      if P <> NIL then repeat
          write('Anfuegen wen:');readln(E.ident);
          if E.ident<>'' then NeuerAst(P,E)
        until E.ident=''
      else begin
        writeln('Moegliche Anfuegeknoten sind:');Zweiginliste(B,List);
        DruckenknotenListe(List);LoeschenknotenListe(List) end
  until E.ident='Ende';     (* Einlesen beendet *)

  repeat                       (* Begin Auswertung *)
    write('Informationen zu:');readln(E.ident);
    Findeknoten(B,E,P);    (* P-gefundener Knoten *)
    if P<> NIL then begin
      write('(V)orgesetzte ! (a)lle/(d)irekte Unterstellte :');
      read(kbd,wahl);writeln(wahl);
      case wahl of
       'V','v':RueckinListe(P,List);
       'A','a':ZweiginListe(P,List);
       'D','d':UnterinListe(P,List) end;
      DruckenKnotenListe(List);writeln;
      LoeschenKnotenListe(List);end
    else write('Knoten mit Eintrag:  ,E.ident,' nicht enthalten');
  until E.ident='';
  release(Haeptop);
end.
```

*Bild 53 Programm zur Baumanalyse*

von der konkreten Typvereinbarung für den Typbezeichner *Eintragtyp* abhängig. Seine Vereinbarung muß verständlicherweise vor allen anderen Komponenten der Lösung erfolgen. Im Beispiel wird diesem Typ ein Record-Charakter vermittelt, wobei im Programm aber lediglich das Feld *ident* genutzt wird. Anhand dieses Feldes wird in der Funktion *gleich* auch die Entscheidung getroffen, ob zwei Knoten identisch sind. Gleichfalls wird auch nur dieses Feld in der Prozedur *DruckeKnoten* zur Ausgabe angewiesen. Diese Ausgabe auf dem Bildschirm erscheint jeweils linksbündig in einem 20stelligen Bereich. Der Anweisungsteil des Beispielprogrammes selbst kann in zwei Teile untergliedert werden. Im ersten Teil wird im Hauptspeicher der Baum aufgebaut und im zweiten Teil erfolgt die Auswertung der im Baum enthaltenen Informationen.

Der Aufbau des Baumes, der in der Praxis aus einer Datei heraus erfolgen wird, ist hier schrittweise programmiert. Zuerst wird nach dem Namen für den Kopfknoten gefragt und der Baum mit diesem Kopfknoten eröffnet. Die Variable B dient im ganzen folgendem Programm als Zeiger auf diesen Kopf des Baumes und wird nicht wieder verändert. Anschließend wird der Aufbau der einzelnen Verzweigungen organisiert. Dazu wird jeweils zuerst nach dem Knoten gefragt, an den neue Knoten angefügt werden sollen. Ist dieser Knoten vorhanden, so können an diesen beliebig viele Verzweigungen *angehängt* werden. Existiert der Knoten, an den angefügt werden soll, dagegen nicht, so wird der Bediener über alle bereits im Baum aufgenommenen Knoten informiert. (Nur an bereits aufgenommene Knoten des Baumes können weitere *angehängt* werden.)

Nachdem der Baum komplett aufgebaut wurde, der Nutzer die Frage nach dem nächsten Anfügeknoten also mit *Ende* beantwortet hat, beginnt die Auswertung der Baumstruktur. Dazu wird jeweils zuerst gefragt, zu welchem Knoten Informationen ermittelt werden sollen. Dieser Knoten wird im Baum gesucht, die Variable P dient dabei als Zeiger. Wenn der Knoten gefunden wurde, enthält dieser Zeiger nach Beendigung der Prozedur *FindeKnoten* einen von NIL verschiedenen Wert. In diesem Fall werden drei verschiedene Recherchen angeboten. Die vom Bediener gewünschte Analyse wird durchgeführt und unabhängig davon, welche es war, enthält die Variable *List* den Zeiger auf die Ergebnisliste der konkreten Recherche. Diese Liste muß jetzt nur noch durch die Prozedur *DruckeKnotenListe* ausgegeben werden. Nach Ausgabe der Liste auf den Bildschirm wird sie gelöscht, und die nächste Recherche zu einem neuen Knoten kann beginnen.

Dieses Beispielprogramm soll lediglich die Nutzung der entwickelten Komponenten verdeutlichen. Daher wurde es auch nur mit einem Minimum an Dialogführung programmiert. Gerade der Gestaltung des Nutzerdialogs muß in konkreten Programmen natürlich sehr viel mehr Gewicht beigemessen werden.

Zum Abschluß soll noch auf eine weitere mögliche Ausbaustufe der vorgestellten Baumstruktur hingewiesen werden. Bisher verwies ein Knoten auf mehrere Nachfolger, aber nur auf einen Vorgänger. Anstelle des direkten Vorgängerverweises ist aber auch eine Verweisliste für die Knotenstruktur möglich. So erhalten wir eine neue Qualität, denn jeder Knoten kann jetzt eine beliebige Anzahl von Vorgängern und Nachfolgern haben. Die Vielfalt solcher Datenstrukturen erweitert sich erheblich. Aus den Knotenbäumen werden Knotennetze (Bild 54). Die Vorstellung von Prozeduren und Funktionen zum Aufbau und zur Analyse solcher Knotennetze erübrigt sich. Sie können vom interessierten Leser leicht durch kleine Änderungen in den Listings der Bilder 50, 51 und 52 erarbeitet werden.

## Texteditor

Nachdem wir uns in den Teilen 4 und 5 ausreichend mit den Pointern vertraut gemacht haben, werden wir sie in der im weiteren vorzustellenden Lösung, einem kleinen Texteditor, ob ihrer Effektivität des öfteren wiederfinden. Mit Editoren hat praktisch jeder zu tun, der irgendwie mit einem Rechner arbeitet. Die Anzahl der vorhandenen Editoren scheint fast unbegrenzt. Und trotzdem gibt es viele Programmierer, die selbst bereits mehrere Editoren geschrieben haben. Warum ist das so? Jeder Programmierer beginnt seine Arbeit am Rechner mit einem, meist rein zufällig ausgewählten, Editor. Diesen beginnt er zu beherrschen, und seine Grundvorstellung von einem Editor wird geprägt. Nun sind aber Software-Entwickler die kritischsten Nutzer von Software überhaupt, und dieser erste Editor wird irgendwann irgendwelchen Wünschen nicht mehr genügen. Es beginnt die Suche nach dem optimalen Editor, den es aber offensichtlich nicht gibt. Das vorläufige Ende dieser Suche bei einem Programmierer ist eigentlich selbstverständlich: Er schreibt sich seinen eigenen Texteditor. Und dieser realisiert dann meist den Grundumfang des zuerst benutzten Editors, erweitert um die speziellen Wünsche des jeweiligen Software-Entwicklers.

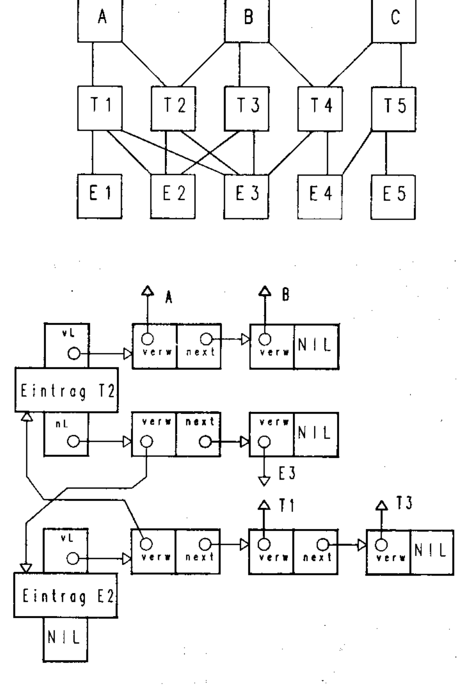



*Bild 54 Vernetzte Knoten*

Es gibt aber noch einen zweiten Grund, sich selbst einen Editor zu schreiben. Man braucht ihn als Quelltext, um ihn in eigene Programme einbinden zu können. Da dieses Problem nicht neu ist, existieren selbstverständlich Tool-Boxen zu diesem Thema. Aber diese meist erst für 16-Bit-Rechner angebotenen Lösungen haben oft eine Quelltextgröße von mehreren tausend Zeilen. Dieser Umfang resultiert meiner Meinung nach aus einem unnötigen Perfektionismus. Es erscheint daher einfacher, einen eigenen, kleinen Editor zu schreiben, als einen dieser *Riesen* zu kürzen und für eine Nutzung auf 8-Bit-Technik umzuschreiben.
Nun zur konkreten Lösung. Als erstes gilt es zu definieren, was ein Text ist, und wie er im Programm behandelt werden soll. In der im folgenden vorgestellten Lösung werden die Zeilen in Form einer dynamischen Listenstruktur gehalten. Die Zeilen stellen dabei ihrerseits Turbo-Pascal-Strings dar. Die maximale Länge der einzelnen Zeilen ist dabei für den gesamten Text gleich, und auf 100 Zeichen begrenzt. Die Anzahl der Zeilen, oder auch die Länge der Textzeilenliste, wollen wir durch den Hauptspeicher begrenzt lassen. Das heißt, ein Retten von Textteilen während des Editierens auf ein Massenspeichermedium ist in dieser einfachen Lösung nicht vorgesehen. Damit ist die Nutzbarkeit auf relativ kleine Texte eingeschränkt.
Die Unterscheidung und Identifikation der Texte erfolgt stets durch einen Zeiger auf ihre erste Zeile, das heißt auf das jeweils erste Element der Textzeilenliste. Auf Grundlage dieser Definition ist es dann auch möglich, mehrere Texte gleichzeitig im Hauptspeicher zu halten und wechselseitig mit den gleichen Funktionen oder Prozeduren zu behandeln. Um dies zu ermöglichen, wird jeder Teilkomponente der Lösung stets über einen Variablenparameter dieser Textkopf übermittelt.
Die einzelnen Komponenten der Lösung sind zur Erläuterung in einen Programmrahmen zusammengefaßt worden (Bild 56). Die Typvereinbarungen am Beginn des Programmes definieren die oben besprochene und zu behandelnde Listenstruktur. Die Variable Textkopf wird als Zeiger auf diese Liste genutzt. Die Anpassung an die konkrete Hardware wird, wie bereits des öfteren beschrieben, über ein entsprechendes Include-File (Bild 55) organisiert, das die benötigten Bildschirmsteuerzeichen und Tastencodierungen beinhaltet. Sie werden im weiteren vorausgesetzt. Weiterhin sind diesmal auch die Eingabeprozedur getkey und die Ausgabe writexy ausgelagert worden, um eine leichtere Anpassung an eine andere Hardware vorzubereiten. Die Funktion *inssatz* und die Prozedur *delsatz* sind für den satzweisen Auf- bzw. Abbau der Textliste im Hauptspeicher verantwortlich. Beiden Konstruktionen wird über den ersten Parameter mitgeteilt, mit welcher Liste sie arbeiten soll, da ja die Arbeit mit mehreren Listen möglich sein soll. Die Prozedur delsatz erhält im zweiten Parameter den Zeiger auf den zu löschenden Satz in der Liste. Der Funktion inssatz wird über die Parameter *vor* und *nach* eindeutig mitgeteilt, an welcher Stelle der Textliste ein neuer Satz eingefügt werden soll. Als Ergebnis von inssatz wird ein Zeiger auf den neuen Satz zurückgegeben.
Es folgen zwei Komponenten, die die Verbindung von den Textdateien auf der Diskette zu der Textliste im Hauptspeicher herstellen. Die Prozedur *Textsichern* ist in der Lage, einen Teil einer Textliste, der durch die Parameter *von* und *bis* eindeutig bestimmt ist, in eine Diskettendatei abzuspeichern. Soll die komplette Liste abgespeichert werden, so muß *von* auf den Textkopf zeigen und in bis muß NIL eingetragen werden. Die Funktion *Textlesen* gestattet die Umkehrung dieses Vorganges, und ist in der Lage, in eine bereits existierende Liste eine Diskettendatei einzufügen. Die genaue Stelle in der Liste ist durch die Parameter *vor* und *nach* bestimmt. Soll an den Anfang einer Liste herangelesen werden, so ist in vor der Zeigerwert NIL und in nach der Textkopf einzutragen. Für den Neuaufbau einer Textliste ist sowohl in vor wie auch in nach ein NIL anzugeben. Die Variable Textkopf wird von der Funktion in jedem Fall berichtigt zurückgegeben. Zur Bestimmung der Bildschirmzeile, in der die Fragen nach den Dateinamen erfolgen sollen, wird beiden Komponenten über y eine Zeilennummer übergeben.
Nun folgt mit der Funktion *Textedit* der Editor selbst. Ihm werden drei Parameter übergeben. Die ersten beiden bestimmen den Bildschirmbereich, in dem der Editiervorgang er-

```
   (* Vereinbarungen fuer  PC 1715   unter CPA  *)
const
  Nop        =   #00;    (*  Quasileeres Char       *)
  Enter      =   #13;    (*  Enter-Taste            *)
  Escape     =   #27;    (*  Escape-Taste           *)
  kursortief =   #24;    (*  Kursortaste tief       *)
  kursorhoch =   #05;    (*  Kursortaste hoch       *)
  kursorrechs=   #04;    (*  Kursortaste rechts     *)
  kursorlinks=   #08;    (*  Kursortaste links      *)
  pageup     =  #$12;    (*  Page up                *)
  pagedown   =  #$03;    (*  Page down              *)
  tablinks   =  #$01;    (*  Wort links             *)
  tabrechs   =  #$06;    (*  Wort rechts            *)
  DEL        =  #$7F;    (*  DEL - Taste            *)
  CE         =   #24;    (*  CE  - Taste            *)
  STaste     =    #2;    (*  S   - Taste            *)
  kursoron   =  #$82;    (*  Kursor ein             *)
  kursoroff  =  #$83;    (*  Kursor aus             *)
  normvideo  =  #$84;    (*  normale Darstellung    *)
  invsvideo  =  #$85;    (*  inverse Darstellung    *)
  intsvideo  =  #$86;    (*  intensive Darstellung *)
  ivitvideo  =  #$87;    (*  invers und intensiv    *)
  delrest    =  #$16;(* Zeile ab Position loeschen *)
  delline    =  #$18;(* ganze Zeile loeschen       *)
  clrscrc    =  #$0C;(* Loeschen ganzen Bildschirm *)
  cleardown  =  #$14;(* Loeschen ab Kursorposition *)

procedure getkey(var c:char);
   begin  read(kbd,c)    end;
procedure writexy(x,y:byte;s:string80);
   begin  gotoxy(x,y);   write(s)  end;
```

*Bild 55 Vereinbarungen für den PC 1715*

```
program editor;
type string80   = string[80];
     zeilentyp  = string[100];
     pointer    =^pointr;
     pointr     = record s:zeilentyp;v,n:pointer end;
var  Textkopf,Heaptop:pointer;

{$I cons1715.inc}

function exist(s:string80):boolean;
  var d:file;
  begin assign(d,s);{$I-};reset(d);{$I+};
    if ioresult<>0 then exist:=false else exist:=true;
    close(d)
  end;

procedure delsatz(var Textkopf:pointer;aktu:pointer);
  begin
    if aktu^.v<>NIL then aktu^.v^.n:=aktu^.n
                    else Textkopf  :=aktu^.n;
    if aktu^.n  <>NIL then aktu^.n^.v:=aktu^.v;
    dispose(aktu) end;
function
inssatz(var Textkopf:pointer;vor,nach:pointer):pointer;
  var tem:pointer;
  begin
    if ((maxavail<85) and (maxavail>0))  then
      begin inssatz:=NIL;exit end;
    new(tem);tem^.n:=nach;tem^.v:=vor;tem^.s:='';
    if vor<>NIL  then  vor^.n:=tem else Textkopf:=tem;
    if nach<>NIL then nach^.v:=tem;
    inssatz:=tem end;

procedure Textsichern(y:byte;von,bis:pointer);
   var s,Dateiname:string80;
       d:text;
   begin if von<>NIL then begin
     writexy(55,y,'Datei-Sichern:'); read(Dateiname);
     if Dateiname<>'' then begin
       assign(d,Dateiname); rewrite(d);
       while((von<>NIL) and (von<>bis)) do
         begin writeln(d,von^.s); von:=von^.n end;
       close(d);
   end end end;

function Textlesen
   (y:byte;var Textkopf:pointer;vor,nach:pointer):pointer;
   var Dateiname:string80;
       d:text;
   begin if vor=NIL then Textkopf:=NIL;
     writexy(55,y,'Datei-Laden  :'); read(Dateiname);
     if exist(Dateiname) then begin
       assign(d,Dateiname); reset(d);
       while (not(eof(d))and((MaxAvail>85)or(Maxavail<0)))do
         begin
           vor:=inssatz(Textkopf,vor,nach); readln(d,vor^.s)
         end;
       if not(eof(d))then {Datei nicht komplett eingelesen!};
       close(d) end;
     Textlesen:=Textkopf
   end;
```

```
function Textedit(y0,ye:byte;Textkopf:pointer):pointer;
  var x,y,i:byte;c:char;znr:integer;
      Tem,aktu    :pointer;
  procedure schr(y:byte;aktu:pointer);
    begin writexy(1,y,aktu^.s+delrest) end;
  procedure schreiben(y:byte;poi:pointer);
    begin while (poi<>NIL) and (y<=ye) do
      begin schr(y,poi); y:=y+1; poi:=poi^.n end;
      if y<=ye then writexy(1,y,delline);
    end;
  procedure bild(y:byte;poi:pointer);
    begin while (y<>y0) and (poi^.v<>NIL) do
      begin y:=y-1; poi:=poi^.v end;
      if (y<>y0) then writexy(1,y-1,delline);
      schreiben(y,poi)
    end;
  function worttest:boolean;
    const fest :set of char = ['A'..'Z','a'..'z','0'..'9'];
    begin
      worttest:=((aktu^.s[x]in fest)and not(aktu^.s[x-1]in fest))
    end;
  procedure zeilv; begin aktu:=aktu^.v ;znr:=znr-1 end;
  procedure zeilp; begin aktu:=aktu^.n ;znr:=znr+1 end;

  begin     (* Begin des  Hauptblockes der Funktion Textedit *)
    write(kursoroff);
    if (Textkopf=NIL) then Textkopf:= inssatz(Textkopf,NIL,NIL);
    aktu:=Textkopf; znr:=1; x:=1; y:=y0; c:=#0;
    repeat for i:=y0 to ye+1 do writexy(1,i,delline);
     bild(y,aktu);
     repeat  gotoxy(35,ye+1);
       write(znr:4,' Spalte:',x:3,' DEZ=',integer(c):4,kursoron);
       gotoxy(x,y);getkey(c); write(kursoroff);
       case c of
        ^E,kursorhoch :if (aktu^.v<>NIL) then begin zeilv;
                        if y>y0 then y:=y-1
                                else schreiben(y,aktu) end;
        ^X,kursortief :if (aktu^.n<>NIL) then begin zeilp;
                       if y<ye then y:=y+1 else bild(y,aktu) end;
        ^D,kursorrechs:if x<80 then x:=x+1;
        ^S,kursorlinks:if x>1  then x:=x-1;
        ^A,Tablinks:if x>1 then
                     repeat x:=x-1 until worttest or(x=1)
                   else if (aktu^.v<>NIL) then begin zeilv;
                     x:=length(aktu^.v^.s)+1;
                   if y>y0 then y:=y-1 else schreiben(y,aktu)end;
        ^F,Tabrechs:if (x<=length(aktu^.s)) then repeat
                    x:=x+1 until worttest or (x>length(aktu^.s))
                   else if aktu^.n<>NIL then begin zeilp; x:=1;
                    if y<ye then y:=y+1  else bild(y,aktu)  end;
        pageup  :begin for i:=y0 to ye-1 do begin
                  if aktu^.v<>NIL then begin zeilv;
                    if y>y0 then y:=y-1 end end;
                  if y=y0 then schreiben(y,aktu)  end;
        pagedown:begin for i:=y0 to ye-1 do begin
                  if aktu^.n<>NIL then begin
                    zeilp;if y<ye then y:=y+1 end end;
                  if y=ye then bild(y,aktu) end;
        ^W        :begin if ((y<ye) and (aktu^.n<>NIL)) then y:=y+1
                  else if (aktu^.v<>NIL) then zeilv;
                  bild(y,aktu) end;
        ^Z        :begin if ((y>y0) and (aktu^.v<>NIL)) then y:=y-1
                  else if (aktu^.n<>NIL) then zeilp;
                  bild(y,aktu) end;
        enter     :begin znr:=znr+1;
                     Aktu:=inssatz(Textkopf,Aktu,Aktu^.n);
                     if x=1 then begin
                       aktu^.s := aktu^.v^.s; aktu^.v^.s:='' end
                     else begin
                       if (x)<length(aktu^.v^.s) then begin
                         Aktu^.s   := copy(Aktu^.v^.s,x,78);
                         Aktu^.v^.s:= copy(Aktu^.v^.s,1,x-1) end;
                       x:=1 end;
                     if y<ye then begin y:=y+1;
                       schreiben(y-1,Aktu^.v) end
                     else bild(y,Aktu) end;
        DEL       :if x>1 then begin
                     x:=x-1; delete(aktu^.s,x,1);schr(y,aktu) end
                   else if Aktu^.v<>NIL then begin
                     x:=length(Aktu^.v^.s)+1;
                     Aktu^.v^.s := Aktu^.v^.s + Aktu^.s;
                     Tem:=Aktu; zeilv; delsatz(Textkopf,Tem);
                     if y>y0 then y:=y-1;schreiben(y,Aktu) end;
        ^G        :begin delete(aktu^.s,x,1); schr(y,aktu) end;
        ^Y        :begin x:=1;
                     if aktu^.n<>NIL then begin
                       aktu:=aktu^.n; delsatz(Textkopf,aktu^.v);
                       schreiben(y,aktu) end
                     else if aktu^.v<>NIL then begin
                           aktu:=aktu^.v;delsatz(Textkopf,aktu^.n);
                           bild(y,aktu) end
                          else begin aktu^.s:=''; schr(y,aktu) end
                   end;
        ^N        :begin x:=1;aktu:=inssatz(Textkopf,aktu^.v,aktu);
                     schreiben(y,aktu) end;
        ' '..#126:begin
                   while length(aktu^.s)<x do aktu^.s:=aktu^.s+' ';
                   insert(c,aktu^.s,x);schr(y,aktu);
                   if x<80 then x:=x+1 end;
      end; { case }
    until c in [^K,#27];
    writexy(1,ye+1,
'S-ichern R-Lesen D-Sichern&Ende Q-uit J-ump'+delrest+kursoron);
    getkey(c);c:=upcase(c);
    case c of
     'D','S',^D,^S: Textsichern(ye+1,Textkopf,NIL);
     'J'    : begin writexy(65,ye+1,'Jump-to:');readln(i);
                while (i<>znr)and(i<>0) do
                  if (znr<i) then
                    if aktu^.n<>NIL then zeilp else i:=0
                  else if aktu^.v<>NIL then zeilv else i:=0
              end;
     'R'    : Textkopf:=Textlesen(ye+1,Textkopf,aktu,aktu^.n) end;
   until c in ['Q','D','X',^Q,^D,^X,#27];
   Textedit:=Textkopf;
  end; { Textedit }

begin { Hauptprogramm }
  mark(Heaptop);clrscr;
  Textkopf:=Textlesen(24,Textkopf,NIL,NIL);
  Textkopf:=Textedit (1,23,Textkopf);
  release(Heaptop);
end.
```

***Bild 56 Texteditor***

folgen soll. Über den Parameter Textkopf wird, wie bereits beschrieben, die Textzeilenliste übergeben. Soll ein neuer Text eingegeben werden, muß hier NIL stehen. Der Ergebniswert der Funktion Textedit ist ein Zeiger auf die geänderte Liste.

Die lokalen Variablen der Funktion Textedit werden wie folgt genutzt: X und y sind die jeweils aktuelle Bildschirmposition des Kursors, x bestimmt damit auch die aktuelle Änderungspostion im jeweiligen String. Die Zeichenvariable c dient der Tastaturabfrage und damit der Funktionssteuerung. In der Integerzahl *znr* wird die aktuelle Zeilennummer mitgezählt. Mit der Pointer-Variablen *aktu* wird das jeweils in Arbeit stehende Textlistenelement identifiziert. *Tem* ist eine temporär genutzte Variable. Einige wenige Teilaufgaben sind als lokale Prozeduren oder Funktionen programmiert. Sie haben folgende Aufgaben: Die Prozedur *schr* schreibt die Zeile des mit aktu identifizierten Listenelementes auf die mit y bestimmte Zeile des Bildschirmes und löscht den Rest der Zeile. Der Prozedur *schreiben* wird die aktuelle Bildschirmzeile und das aktuelle Listenelement übergeben. Von dieser Zeile an aktualisiert die Prozedur das Bildschirmaussehen anhand der Textliste. Die Prozedur *bild* erhält als Parameter die gleichen Informationen. Sie aktualisiert aber den gesamten Bildschrim. Dem wortweisen Springen beim Editieren dient die Funktion *worttest*. Durch die Prozeduren *zeilv* und *zeilp* im Funktionskörper wird die starre Kopplung zwischen der zu bearbeitenden Textliste und dem Zeilenzähler *znr* generell garantiert.

Im Anweisungsteil der Funktion wird nun präzise definiert, welche Tastatureingabe welche Reaktion des Editors hervorruft. Die schwierigste Aufgabe dabei ist die ständige Übereinstimmung von Bildschirmaussehen und Textinhalt. Das eigentliche Editieren erfolgt in der inneren Repeat-until-Schleife, die fast nur aus einer Case-Verzweigung besteht. Die äußere Schleife beinhaltet zusätzliche komplexere Funktionen und kann noch viel weiter ausgebaut werden. Das hier organisierte Abspeichern (∧KD) und Lesen (∧KR) von Diskettendateien ist das notwendige Minimum. Das zusätzlich aufgenommene Springen (∧KJ) auf eine konkrete Zeile soll die Möglichkeit von Erweiterungen nur andeuten.

*Schluß*

**✉ KONTAKT ☎**

Manfred Zander, Schäferstraße 11, Dresden, 8010